Report No. UIUCDCS-R-79-964

UILU-ENG 79 1709

# SCHEDULING AND SIMULATION OF COMPUTATION GRAPHS FOR MULTIPLE RESOURCE COMPUTERS

by

John Charles Wawrzynek

March 1979

NSF-OCA-MCS76-81686-000040

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-79-964

# SCHEDULING AND SIMULATION OF COMPUTATION GRAPHS FOR MULTIPLE RESOURCE COMPUTERS

by

John Charles Wawrzynek

March 1979

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS76-81686 and the Department of Computer Science, and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering, March 1979.

ACKNOWLEDGMENT

I wish to express my sincere thanks to Duncan Lawrie for his guidance thoughout the course of my master's work. I would also like to thank Frank Panzica, for his many suggestions and help with debugging FAPGEN, and Bob Montoye for his help with writing the programs.

# TABLE OF CONTENTS

# I. INTRODUCTION

Studies of scientific Fortran programs have revealed some ever present vector operations. An attempt to improve the performance of computers with respect to these vector operations has inspired the design of several new high performance computers. These machines include pipelined processors as well as parallel machines. While both types have their shortcomings, designs employing a combination of the two seem very effective. One such design is that of the Burroughs Scientific Processor (BSP). The BSP combines parallelism and pipelining by providing a five segment memory-to-memory data pipeline of memory, alignment network, processor, alignment network, and employs parallelism through the use of 17 parallel memories and 16 parallel processors [1].

As with the design of any new machine there exists a need to provide an accurate simulation of its operation to study its effectiveness. Also of practical importance is the design of a control process which can guarantee a high degree of utilization of the machine resources. In this paper we present a scheme for controlling such a machine and in doing so provide the means for a very precise simulation.

An effective control process for a vector machine must begin at a much higher level than that of conventional computers. A project currently underway is directed at analyzing Fortran programs to discover and improve their inherent parallelism [2]. Along with vector operations, the programs analyzed contain a large number of linear recurrence relations. Many fast, efficient algorithms exist for computing linear recurrences [3,4,5].

The problem we set out to solve is to find a means to simulate the execution of this class of algorithms on a machine with a structure similar to the BSP. Such a simulation can aid in the solution of questions of machine design and the efficiency of these algorithms. Since the algorithms can be decomposed into sections of vector type operations, the results apply to vector operations as well.

Although this work is directed primarily towards the control and simulation of BSP type machines, the theory included in our approach is generalized to include any machine consisting of a collection of interconnected resources. The main theme is to schedule the operations in the resources so as to overlap their activities and keep each as busy as possible throughout the course of a computation.

Chapter II discusses the structures of our assumed machine, the algorithms, and a chart used to represent the schedule of activities in the machine. We also present some relationships amoung the three and some basic theorems. In Chapter III we present our approach to the problem and make some remarks about machine design. A PL/1 program called FAPGEN has been developed to implement our approach. It is discussed in Chapter IV. Chapter V has some experimental results of our methods and some possible extentions and problems. The appendix includes a user's guide for FAPGEN.

# II. BACKGROUND

This chapter attempts to redefine the problem in terms of the structures of the machine, the computational algorithm it must support, and the schedule of operations. With these representations some basic ideas become obvious. Also these representations provide a means to study the relationships between machines, algorithms, and solutions, while providing a basis for further investigation.

## A. The Machine

A general model for machines can be described without any concern for the behavior of each device or box in the system. This removes the need to be concerned with the operation of the actual function being preformed by the system, thus focusing attention on its structure. With this in mind, the machine can be defined. A machine is a collection of boxes connected by paths which provide a means of passing data, and is denoted by M. Each box can contain one or more registers at its output or input. Global control is assumed. Figure 1 shows an example of a machine. In this case only a subset of all the possible interconnections are utilized. In general, for a machine of n boxes, $n^2$ interconnections are possible.

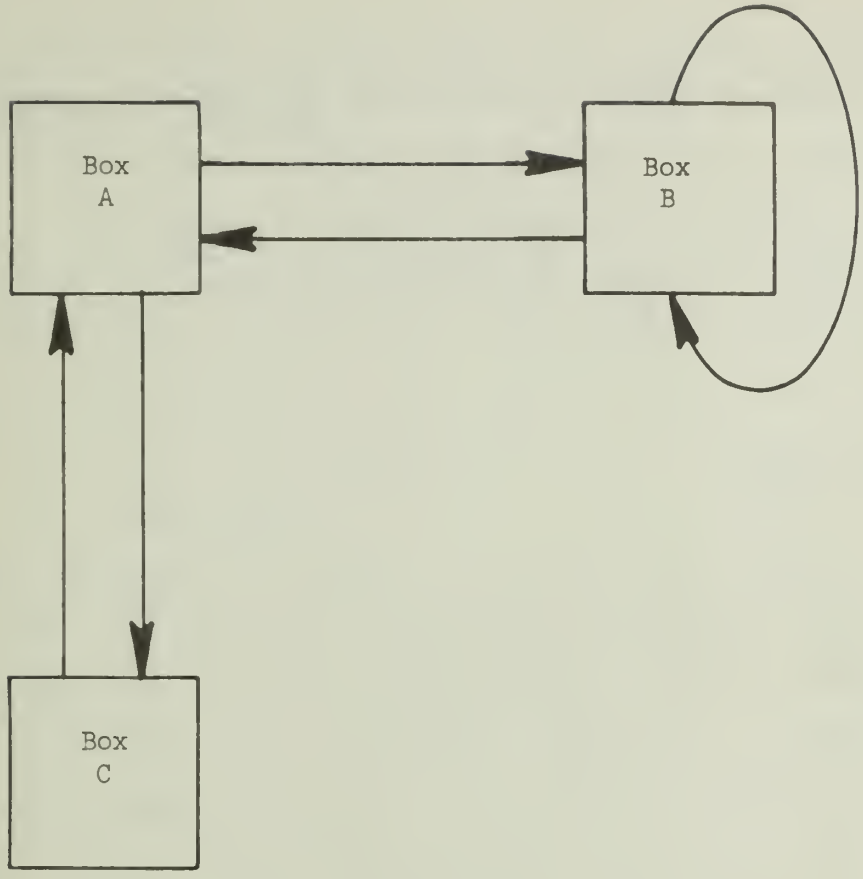



Figure 1. Machine M1

Figure 2 shows a machine of considerable practical importance. This is sort of a circular pipeline and is the structure of the Burroughs Scientific Processor. In this case, and in general, the interconnections are multiple data paths.

B. The Graph

For each machine, there exists a class of computation algorithms. Each is represented by a directed graph, G. Every node in the graph represents a unit operation to be performed within one box of the machine. Again there is no concern for the actual function performed within the box, only the time is important. Each node in the graph must be qualified by a minimum of three things: 1. An identification of the box in which the operation is to be performed. 2. The time until the data is ready at the output of the box. 3. The time until the box can be used again. This first time is known as the data access time, $T_a$. The second time is known as the busy time, or $T_b$. For example, for core memory, $T_a$ might be the access time and $T_b$ the busy time, where $T_a < T_b$. Thus data might be available in 500 ns, but the memory might not be ready for a new operation for another 500 ns, yielding $T_a$ = 500 ns and $T_b$ = 1000 ns. An example where the converse is true is in the case of a pipelined resource, where it is possible to

feed operands into the pipe before the results of the last operation are available.

If two nodes are connected by an arrow, they are said to be adjacent. If the arrow connecting two adjacent nodes, x and y, begins at node x and terminates at node y, then node x is said to be a predecessor of node y, and node y is said to be a successor of node x. Also, we say that node y is data dependent on node x, and denote this by x->y. If x->y, then the operation represented by node y cannot be performed until the data generated by node x has been computed. If nodes x and y represent operations by the same box, then the time between the start of node x and the start of node y is at least the busy time associated with node x. However, if the two nodes represent operations by different boxes, then the time from the start of node x to the start of node y must be at least the access time associated with node x.

Groups of nodes can be enclosed by a bracket to represent iteration, and a corresponding number, N, to identify the number of iterations to be performed (see Figure 4).

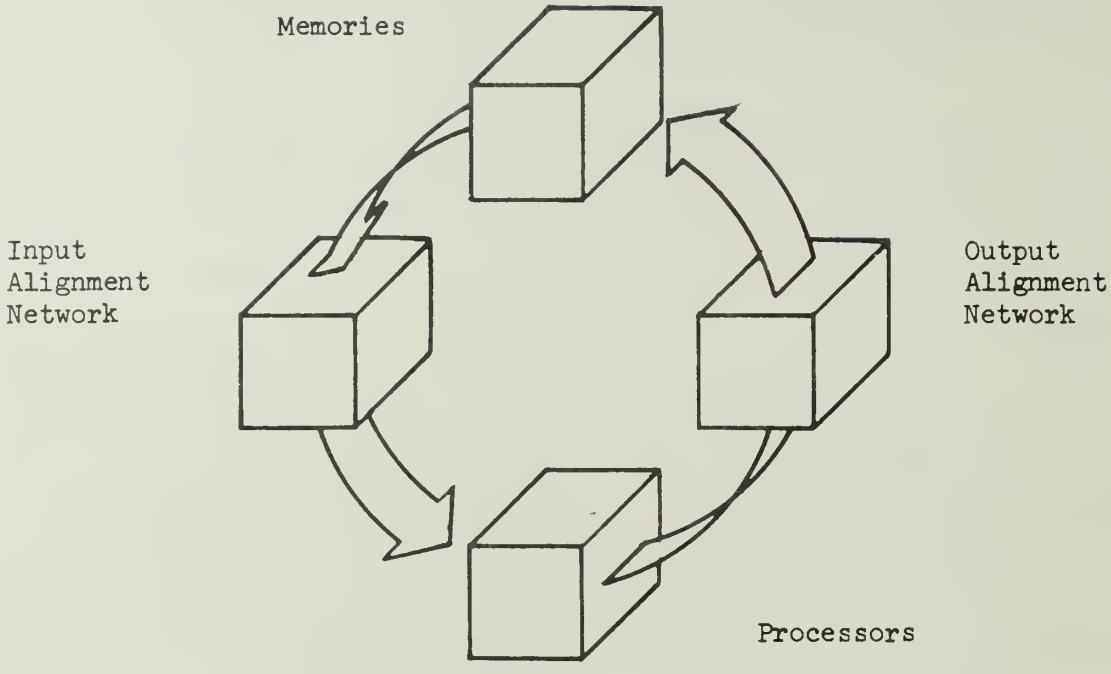



Figure 2. Structure of the BSP

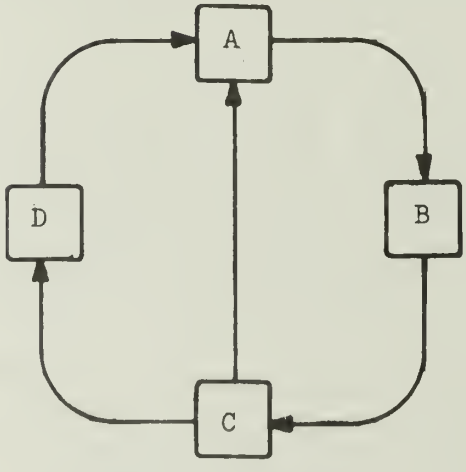



Figure 3. Machine M3

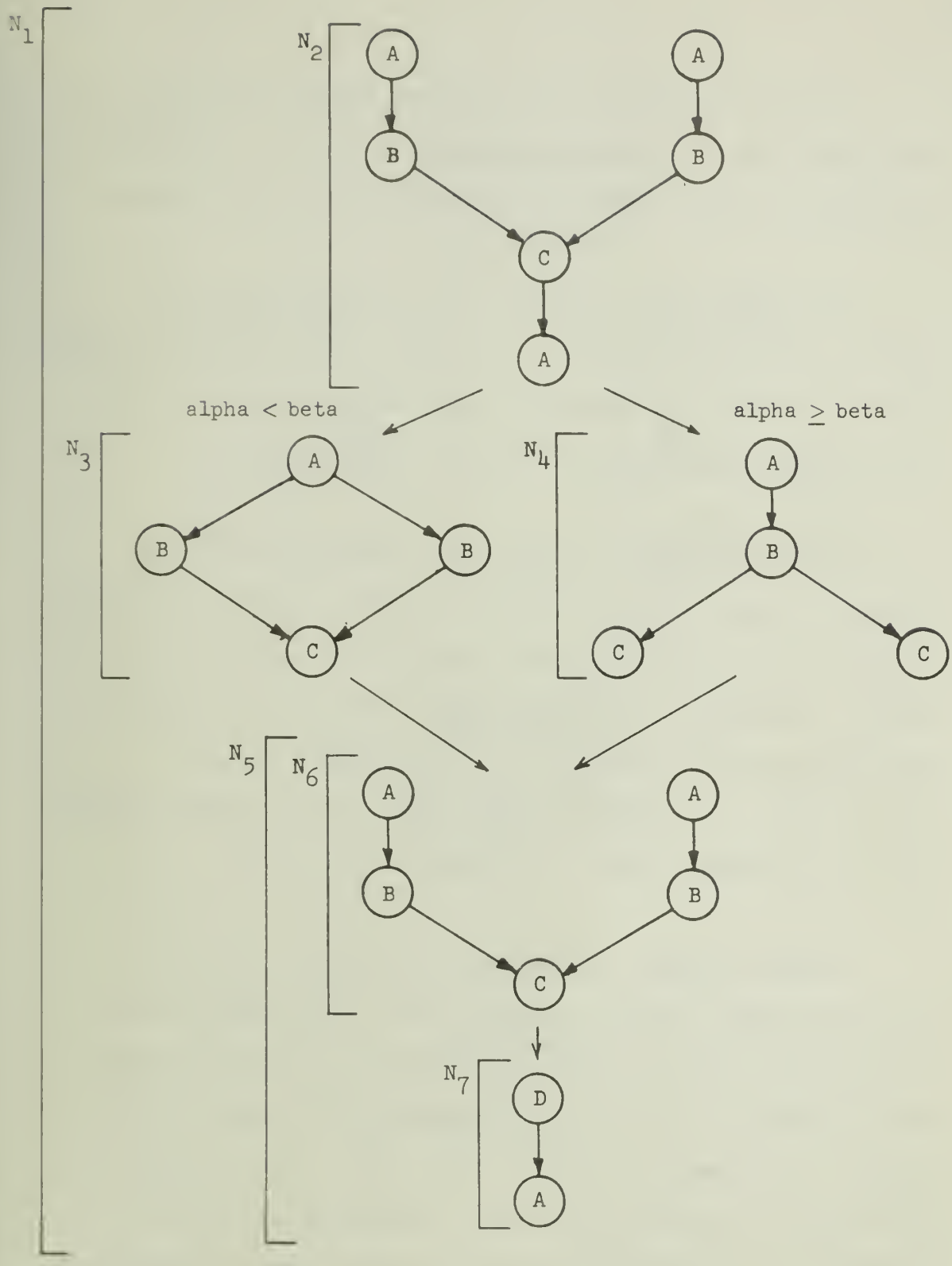



Figure 4. Computation Graph for Machine M3

An arrow pointing in the upward direction within an iteration loop represents a dependence between succesive iterations of that loop. Iteration loops can be nested.

With these basic constructs, any algorithm for a machine can be represented. Figures 3 and 4 show an example of a machine and one possible computation graph G. The letter within each node identifies a box in M.

It is important to note that each of the IF-free groups of nodes, or any section thereof, could be replaced by some higher level operator whose function is that of the nodes it replaces. This concept is identical to that of microinstructions combining to form machine level instructions. As will be seen later, the innermost nested loop is a convient level to decompose the graph.

Figure 5 shows the computation graph with the innermost loops replaced with blocks. Decomposition of algorithms in this manner introduces the need to include extra information as to the dependencies between groups of nodes. For example, consider Figure 5. If the operation indicated by a node in the first iteration of block 2 is dependent on the result of a node in the last iteration of block 1, then this would have to be explicit.

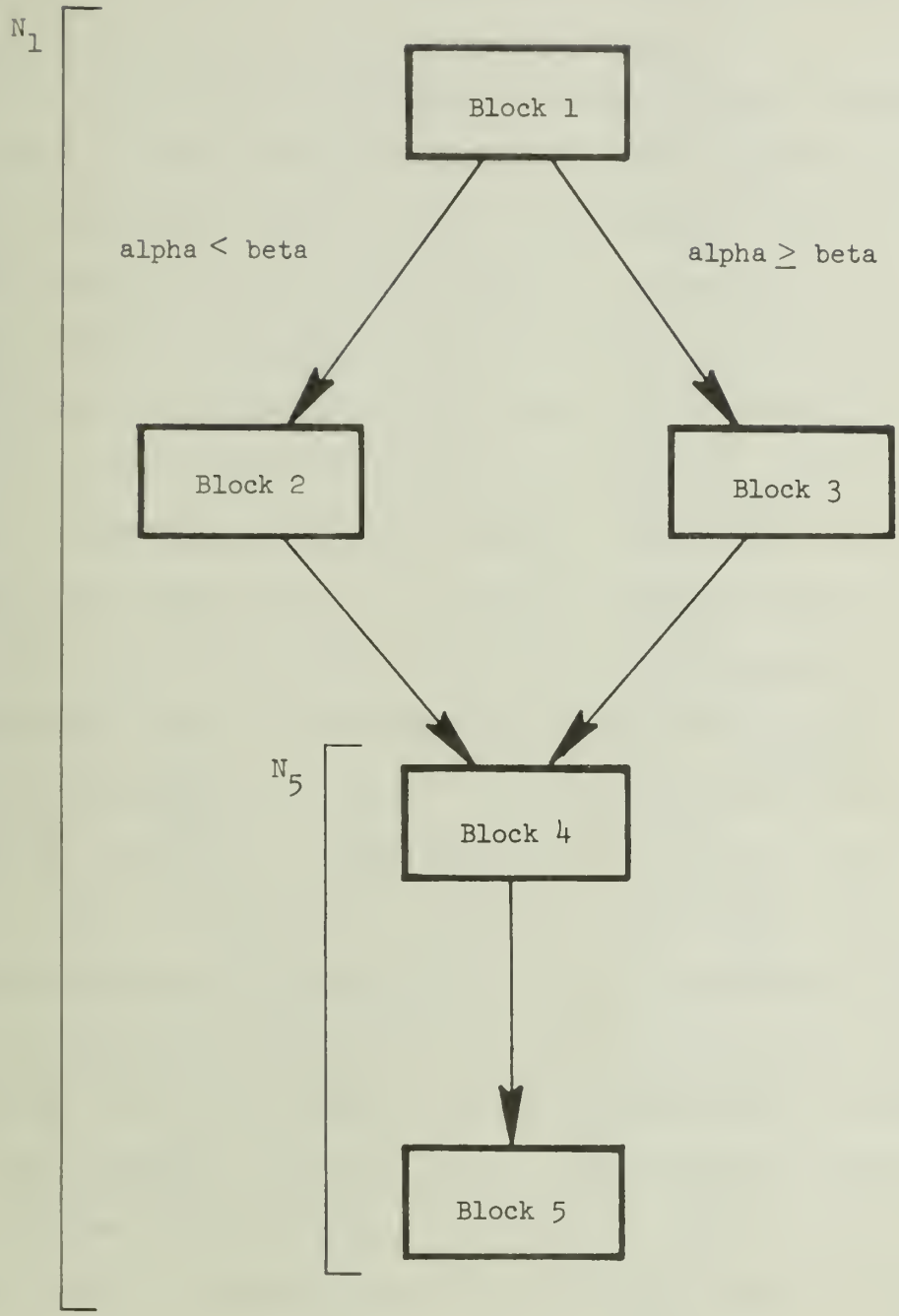
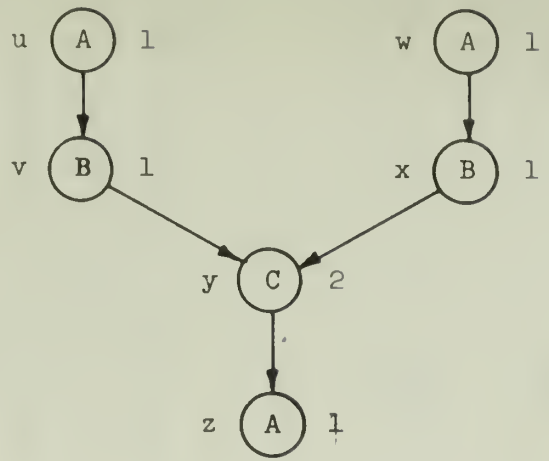



Figure 5. Replacement of Innermost Loops with Blocks

C. The Schedule Chart

Given a machine and an associated computation graph, the problem becomes that of scheduling the boxes of the machine in such a manner as to minimize the time needed to execute the algorithm. The approach will be to treat each block seperatly; the results of the appropriate blocks can later be joined. Again, each block represents an innermost iteration loop. A representation of this scheduling is a device vs. time chart. Figure 6 shows the device vs. time chart for block 1 of G1 with N2 = 7. Here each node in the graph is identified by a character appearing to its left. To the right of the nodes is the number of time units needed for that particular operation. For this example it is assumed that $T_a = T_b$ for all operations. Each row, or rail, in the chart stands for one box in the machine. The vertical lines mark off time divisions. The representation of a node from G on the chart is called an <u>occurrence</u> of that node and corresponds to the scheduling of the operation represented by that node. The occurrence of node x is denoted by $x_i$, where the subscript identifies to which iteration of the graph the occurrence belongs. Every node in one iteration of G has the same subscript. We will assume that the chart represents a sequential execution of the graph, namely, iteration i cannot begin before iteration i-1 has begun.




Box

| | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | $u_1$ | $w_1$ | | $u_2$ | $w_2$ | $z_1$ | $u_3$ | $w_3$ | $z_2$ | $u_4$ | $w_4$ | $z_3$ | $u_5$ | $w_5$ | $z_4$ | $u_6$ | $w_6$ | $z_5$ | $u_7$ | $w_7$ | $z_6$ | | | |
| B | | $v_1$ | $x_1$ | | $v_2$ | $x_2$ | | $v_3$ | $x_3$ | | $v_4$ | $x_4$ | | $v_5$ | $x_5$ | | $v_6$ | $x_6$ | | $v_7$ | $x_7$ | | | |
| C | | | | $y_1$ | $y_1$ | | $y_2$ | $y_2$ | | $y_3$ | $y_3$ | | $y_4$ | $y_4$ | | $y_5$ | $y_5$ | | $y_6$ | $y_6$ | | $y_7$ | $y_7$ | |

time →

Figure 6. Device vs. Time Chart for Block 1 of G1 with $N_2 = 7$

Although the chart for G1 is not unique, after the first occurrence of node x, it is the optimal solution. This is easily proved by the fact that from this point on box A is always busy. The execution of this block is said to be "bound" by box A.

This optimal solution could only have been achieved by providing sufficient lookahead in order to prevent "collisions", or conflicts. This can be seen by observing the hole, or blank square, in rail A. Although this may seem like wasted time for the first iteration, thereafter it is utilized by A. Examination of the chart will yield another interesting observation; after an initial startup time, the chart becomes periodic. Then after a number of iterations there is a final transient. The startup time will be referred to as the HEAD, the center periodic section as the BODY, and the final section as the TAIL.

The entire BODY of the chart can be represented by one characteristic section, along with its multiplicity. We will call the characteristic BODY section the window. It is defined as the smallest number of contiguous columns which uniquely identify the BODY of the chart. The length of the window is called the period. With this, we can consider the BODY of the chart as simply being a series of windows.

Each window differs from the others only by a constant in the subscripts. Note that in general, the window for any one BODY is not unique, however, the period is.

The HEAD and TAIL are are special cases of one or more iterations of the representative BODY section. They differ only in the fact that the group of nodes that they represent is a subset of those in a representative BODY section. Therefore it is possible to represent an entire block by one window. This approach is very attractive for it presents a very compact way to store the scheduling of a block. Also, if it is possible to derive one period from the graph, then a solution has been found independent of the number of iterations of the algorithm.

D. Some Basic Theorems

The immediate problem now is to map the graph into one window of the BODY of the chart. In other words, given G, find a window. Eventually the goal will be to find a window in such a way as to minimize the period. Examination of the BODY yields some simple yet powerful theorems which will aid us.

The first theorem is simply a statement about conservation of nodes.

Theorem 1

Given a graph G on machine M and its corresponding schedule chart, within every window if there exists k occurrences of one node from G then there exists exactly k occurrences of every node from G, for k = 1,2,..,N.

Proof

The definition of a window implies that every window is identical, disregarding subscripts, and that it is repeated throughout the chart. Therefore, if we remove the occurrence of a node from any window, it must be removed from every other window, resulting in an incomplete chart. Likewise, if we add the occurrence of a node to one window, it must be added to every other window, resulting in a chart with redundant information.

Q.E.D.

Theorem 1 tells us that within every window each node has the same number of occurrences. In addition to this we know that for every graph there exists a window that includes only one occurrence of each node from the graph. We will consider only this case, that is where k = 1.

Figure 7 will help to demonstrate the next theorem. G2 represents an iterative algorithm. As can be seen in the figure, the process of dividing the BODY of the chart into windows corresponds to partitioning the iterations of G2 as shown. The partitioning of the iterations of G defines the relative subscripts of the node occurrences in the window.

Theorem 2

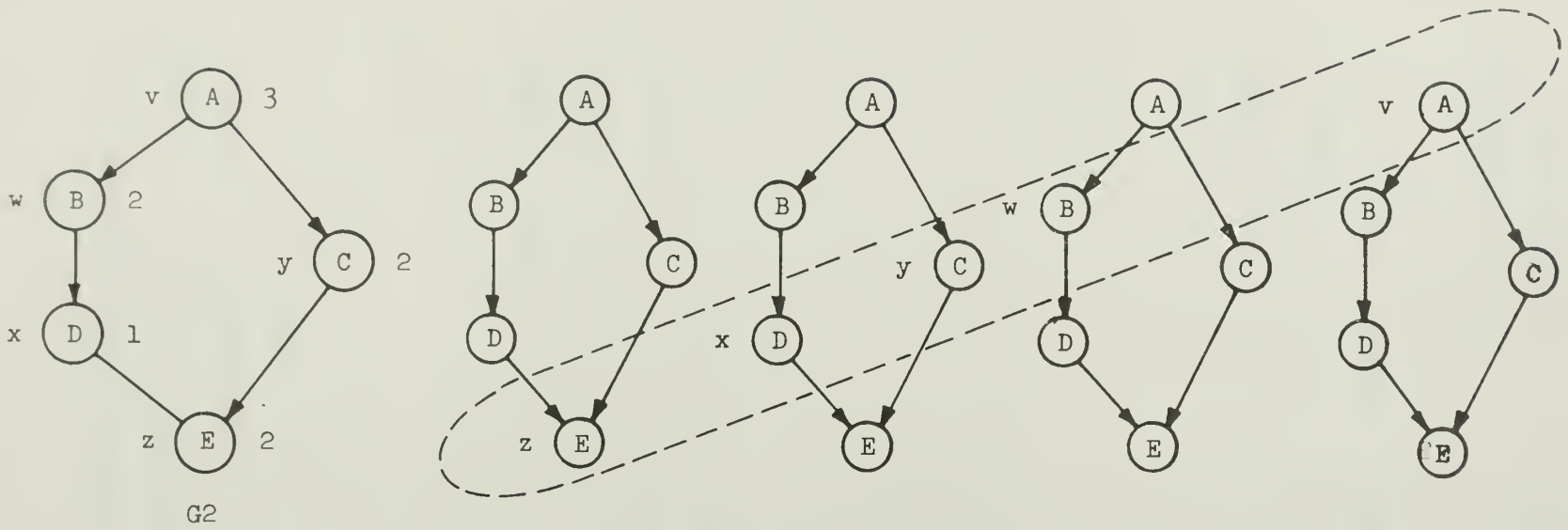
Given occurrences $x_i$ and $y_j$ within one window of any chart, where node y is a successor of node x, then $i \geq j$.

Proof

Assume $x_i$ and $y_j$ where $i < j$ in window k. $y_j$ in window k implies that $x_j$ has already occured. Window k contains $x_i$. Therefore by Theorem 1, $x_j$ must occur outside of window k. This means that $x_j$ precedes $x_i$ in the chart, which contradicts the sequencial ordering of iterations.

Q.E.D.

The last example showed a block which had no dependencies between iterations of the graph. In general there are one or more data dependencies between iterations. For example, in Figure 8, the operation performed by box B




| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | $v_1$ | $v_1$ | $v_1$ | $v_2$ | $v_2$ | $v_2$ | $v_3$ | $v_3$ | $v_3$ | $v_4$ | $v_4$ | $v_4$ | $v_5$ | $v_5$ | $v_5$ | $v_6$ | $v_6$ | $v_6$ | $v_7$ | $v_7$ | $v_7$ |
| B | | | | | $w_1$ | $w_1$ | | $w_2$ | $w_2$ | | $w_3$ | $w_3$ | | $w_4$ | $w_4$ | | $w_5$ | $w_5$ | | $w_6$ | $w_6$ |
| C | | | | | | | $y_1$ | $y_1$ | | $y_2$ | $y_2$ | | $y_3$ | $y_3$ | | $y_4$ | $y_4$ | | $y_5$ | $y_5$ | |
| D | | | | | | | | | $x_1$ | | | $x_2$ | | | $x_3$ | | | $x_4$ | | | $x_5$ |
| E | | | | | | | | | | | $z_1$ | $z_1$ | | $z_2$ | $z_2$ | | $z_3$ | $z_3$ | | $z_4$ | $z_4$ |

Figure 7. Partitioning of Iterations of G2

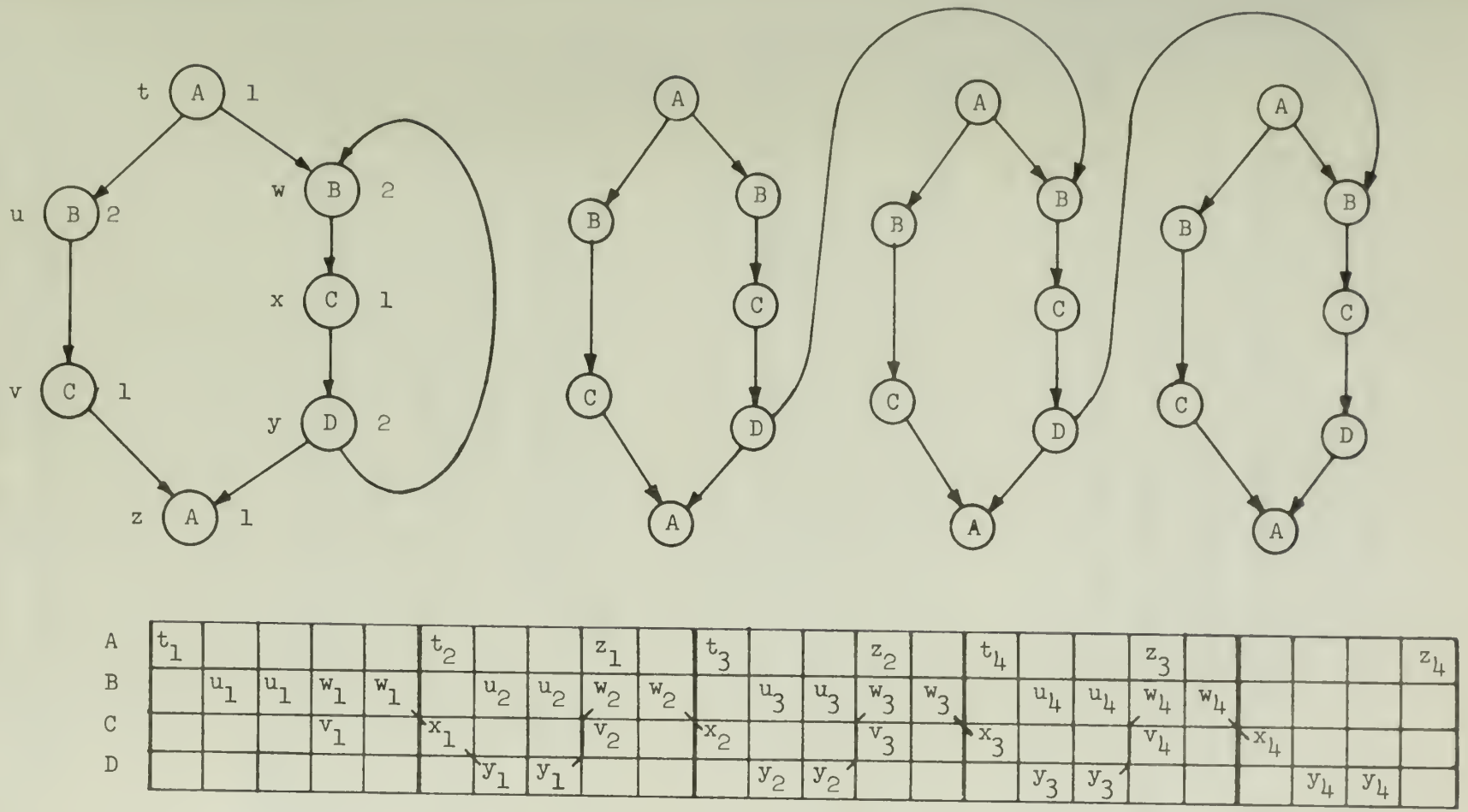



Figure 8. Dependency Between Iterations

at node w depends on the results of the operation performed by box D at node y in the last iteration. This dependency has the effect of limiting the amount that successive iterations can be overlapped on the chart. If the dependency between iterations had not been present, then a "tighter" chart could have been produced and the computation would have been bound by B.

If $x_{i+1}$ is a successor of $y_i$ and $y_i$ is dependent on $x_i$, then there exists at least one dependency cycle in G. In general there is more than one path connecting node y to node x. This means that more that one dependency cycle can be defined by only one dependency between iterations.

Theorem 3 will help in mapping a graph with dependency cycles into a window. First we will define another term. The occurrences $x_i$ and $y_i$ of two nodes x and y are said to be overlapped if $y_i$ occurs during the data access time associated with $x_i$, or vice versa. For the special case where $T_a = T_b$, this means that $x_i$ and $y_i$ have one or more columns in common.

Theorem 3

Given a graph G with one or more dependency cycles, for each cycle, no occurrence of a node in that cycle can be overlapped with the occurrence of any other node in that cycle.

Proof

Assume two nodes, x and y, in a cycle, where $y_i$ is dependent on $x_i$. The last occurrence of node y, $y_n$, implies that $x_n$ has already occurred. Because of the cycle, $x_n$ implies that $y_{n-1}$ has already occurred, and so on for all occurrences of x and y.

Corrollary

Given the occurrences, $x_i$ and $y_j$, of two nodes, x and y, in one window, then j = i or j = i-1.

Proof

Theorem 3 implies an ordering:

$$x_k \rightarrow y_k \rightarrow x_{k+1} \rightarrow y_{k+1} \rightarrow \ . \ . \ . \ \rightarrow x_n \rightarrow y_n.$$

Theorem 1 tells us that each window contains exactly one occurrence of every node in G, therefore a window includes the pair $x_k$, $y_k$ or the the pair $y_k$, $x_{k+1}$.

Q.E.D.

At this point it is possible to define a lower bound on the period of the chart in terms of the computation graph, G.

Theorem 4

Given a graph G, the minimal period is the maximum of two things:

1. The maximum over all cycles of the time needed to traverse the the cycle.
2. The maximum of the total busy times for each box.

Proof

Consider each box needed to execute the graph. Assume we have n nodes in the graph representing an operation on box q. Since there must be an occurrence in the window in row q, for each of the n nodes, the window must at least allow for the sum of the busy times of all n nodes. Next, from Theorem 3, we know that for each cycle none of the occurrences of the nodes contained in that cycle can be overlapped. Therefore the window must be at least large enough for all of the nodes in that cycle with no overlap.

Q.E.D.

## E. Registers and Connections

Several important relationships exist between the machine, the computation graph that it supports, and the schedule chart. The set of connections between boxes and the nature of the registers contained in the boxes define the structure of the machine and have an important effect on the algorithms that the machine supports. If we examine a particular machine we see that the interconnections and the registers limit (make rules about) the form of the computation graph. The use of registers also has a significant effect on the scheduling of the algorithm. This section makes some comments about these relationships.

The most flexible arrangement of interconnections is one in which every box is connected to every other box. For n boxes this requires $n^2$ connections. For most practical cases this scheme is not necessary, in addition to being economically infeasible. At the other extreme, we know that n-1 connections is the minimum number needed to connect every box to at least one other box. If all n-1 connections are bidirectional then it is possible to pass data from any box to any other box. Here it may become necessary to pass data through several boxes on route to their destination.

Obviously, this will slow down the process. For example consider M4 in Figure 9. If an algorithm calls for data to be passed from box E to box A, it would be necessary to route the data through box C then through box B and then finally to box A. The graph shows this. Here the operations associated with nodes v and w are null operations.

In reality it may not be necessary to have the ability to pass data between all of the boxes. A good set of connections would consist of those most commonly used. The less common connections could be achieved in an indirect manner as described above.

If we assume that not every connection needed for the algorithm is present in the machine, we can say that in general the effect of interconnections, or more precisely the lack thereof, increases the time needed to execute the algorithm. Another set of connections, or sneak paths[1], can be added to the machine definition to study this effect.

---

[1]This should not be confused the sneak paths of relay networks.

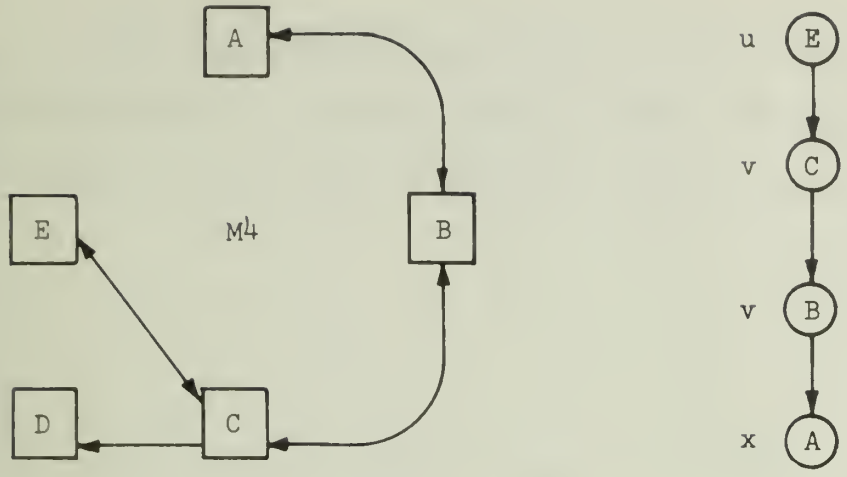



Figure 9. Passing Data Through Boxes

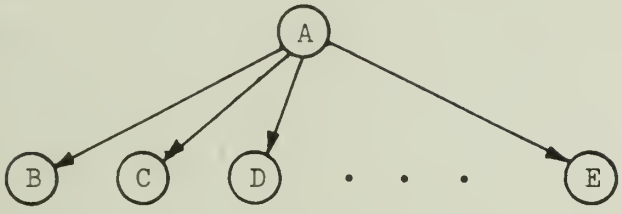



Figure 10a. Fan-out of Data

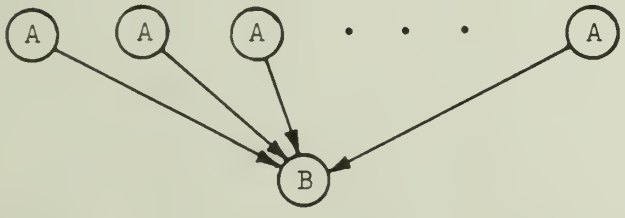



Figure 10b. Fan-in of Data from Same Box

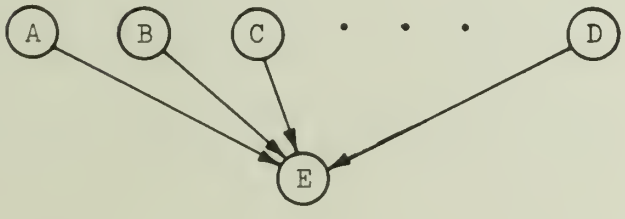



Figure 10c. Fan-in of Data from Different Boxes

The addition of a sneak path between two boxes which are not normally connected speeds up any computation involving those two boxes by eliminating the need to send data through other boxes.

Another important aspect of the machine structure is the presence of registers. Consider the occurrences on a chart of two adjacent nodes x and y. Assume a register exists between the two boxes associated with nodes x and y. After the occurrence of x it is possible to hold its data before the occurrence of y, because the data can remain in the register. The box used by node x could then be used by some other node in the graph, even though node y has not yet finished. However, if no register exists then the occurrence of y must immediately follow that of x. This concept can easily be related back to the graph. If we call the arrow connecting two nodes a link, then a stretchable link is a link which implies that the occurrence of y need not follow immediatly after the occurrence of x. Therefore, if two nodes are connected by a stretchable link then there must exist a register between the two boxes associated with those nodes.

Fan-in and fan-out of data in the graph implies the existence of registers in the machine. In the case of fan-out, registers are needed when fan-out is to different boxes of the machine. Figure 10a shows how the computation graph reflects this. Here box A must contain an output register because some boxes receiving data from A may be busy when box A is ready to send out data.

For fan-in from the same box, as in Figure 10b, f-1 input registers are required for box B. If the fan-in is from different boxes as in Figure 10c, then another possibility exists. Namely, boxes A through D could contain a single output register. Of course, if the operations in boxes A through D all take the same amount of time then no registers are required.

So, if a computation graph contains fan-out or fan-in of data then the machine on which the graph is to be executed must contain at least the appropriate registers to permit the fan-out or fan-in.

# III. PRODUCING SCHEDULE CHARTS

In this chapter we present the solution to the problem of scheduling an algorithm. It has been shown that we can decompose any computation graph into blocks of nodes, where each block is executed iteratively and itself contains no iteration loops. Every node represents a resource request, that is, an operation on a box in the machine.

The first four sections in this chapter discuss methods for producing a schedule for each block, in the form of a window, a HEAD and a TAIL. The last section is concerned with joining the schedule charts for all the blocks to produce the results for the entire algorithm. The implications of these results are twofold. First, this can be considered as a method to determine the time needed to execute a given algorithm. Second, it implies the necessary control store (microprogram) and structure to execute the algorithm.

## A. Wrap-around

In the last chapter we showed how the BODY of the schedule chart for one block of nodes can be divided up into sections called windows. Since the size of the window may

be smaller than the length of one iteration of the graph, each window may contain nodes from several different iterations of the graph. In this manner the window contains overlapped sections of several identical iterations. Our approach to the problem of scheduling one block will be to assume a window size and then try to schedule all the nodes of the graph in the window.

Let us assume that the i-th iteration of the graph starts at the beginning of the i-th window in the chart. We will start at the top of the graph and work down, scheduling the nodes in their respective rows on the chart. As we schedule nodes we assign to them the subscript i until we reach the right window boundary. At this point to continue iteration i of the graph we must cross the right window boundary and extend in window i+1. Since every window is identical this implies that iteration i-1 is being extended in window i. The effect of this is that it is never acually necessary to consider more than one window. When we run out of space in the window i, we simply decrease the subscipts of the occurrences and continue from the left window boundary. In this sense we say that the scheduling of the nodes of a graph are wrapped around in the window.

B. Groups

The discussion in the last section of Chapter II showed that the use of registers in a machine allowed some of the links in the graph to be stretchable. Furthermore, when there is fan-out and fan-in of data, stretchable links must be present, although they may also occur anywhere in the graph. For simplicity we will only deal with graphs where the stretchable links are restricted to the fan-out and fan-in points. In addition, every link at these points is a stretchable one. This provides a nice way to divide the graph into groups. A group is a set of nodes connected by links which are not stretchable. As a consequence of this, there is no fan-out or fan-in within a group. For example consider the graph in Figure 11a. Rectangles are used to indicate the groups. Figure 11b shows these groups mapped into the schedule chart representation. In this form they appear as geometric shapes. Each shape occupies one or more adjacent columns in the chart. Note that the shape implied by the group cannot be altered since all the links involved are unstrechable. With this representation, the scheduling problem becomes that of properly arranging the shapes on the chart.

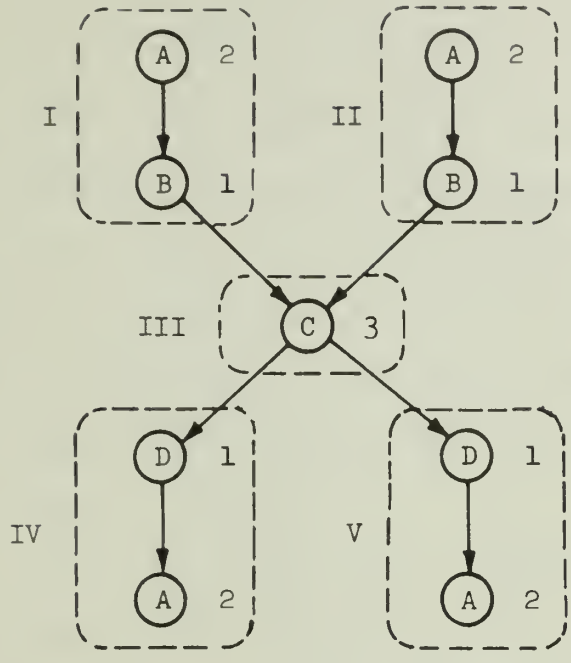



Figure 11a. Division of Graph into Groups of Nodes

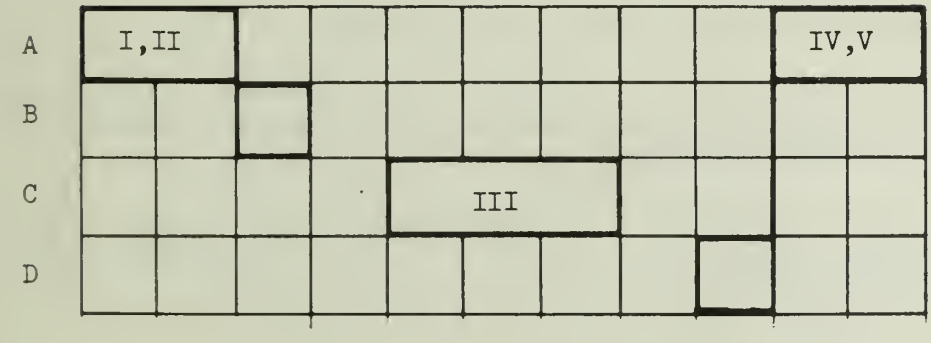



Figure 11b. Groups of Nodes as they Appear on the Chart

C. Procedure for Scheduling a Block

An algorithm has been developed for the generation of a window, given a graph. The theorems along with the idea of groups and some simple heuristics form the basis of the algorithm. Because of our use of a window, we deal with parts of one or more iterations of the graph all within the same window. The basic idea of the algorithm is to estimate the window size and then fill it up with as many nodes as possible from iteration i, then from iteration i-1, etc., until every node has been scheduled. We present the algorithm below, followed by three examples.

1. Identify the dependency cycles.
2. Partition the graph into groups.
3. Use Theorem 4 to determine the minimal window size. This becomes the initial window size and defines the window boundaries.
4. Order the groups for scheduling such that no group is scheduled until all of its predecessors have been scheduled.
5. Schedule all of the groups.
    1. Schedule the first group with its left edge at the left window boundary.
    2. Schedule the remaining groups one at a time.
        1. Try the group immediately to the right of thoses groups already scheduled such that it does not overlap with any of its predecessors.

2. A group can be scheduled only if every node in that group can be scheduled.

3. If a conflict exists in this position then move the group to the right and try again.

4. If the right window boundary is crossed by part or all of the group then wrap the excess around and start at the left window boundary, decreasing the subscript of the nodes involved by one. (It may be useful to visualize the chart as being on the surface of a cylinder whose circumference is equal to the size of the window. Here the right and left window boundaries form a single line.)

5. Continue moving the group to the right and wrapping it around until one of the following occur.

    1. The group fits with no conflicts.

    2. the group has been moved back to its original position.

    3. Theorem 3 is violated. ( no occurrences of nodes in the same dependency cycle can be overlapped.)

3. If the group fits then continue with the next group.

4. If the group has been moved back to its original position, then increase the window size and reschedule all groups.

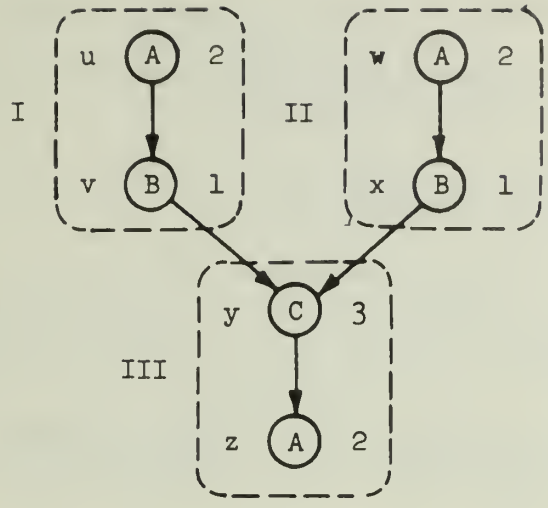
5. If Theorem 3 is violated, then identify the group that contains the occurrences that overlapped with the present group. Disregard the scheduling of all groups scheduled after this one. Schedule the group to the right of its old position. Reschedule all the groups after this one.

The occurrence of the first node in the first group is assigned the subscript i. There after, an occurrence is assigned a subscript equal to the minimum of the subscripts of all its predecessors. If it was necessary to wrap-around then the subscript is one less.

This algorithm has been implemented and has proved to be effective and efficient in most cases of practical interest, although the optimal solution is not guaranteed. Experience has shown that few graphs contain dependency cycles within blocks. Therefore, in favor of simplicity, little effort was made to make the treatment of them efficient.

Now we present the examples. For the first example Figure 12 will be used to demonstrate the algorithm on a simple graph. We will use the notation t(x) to refer to the data access time ($T_a$) for the operation indicated by node x. For these examples we assume that $T_a = T_b$.

1. The graph contains no dependency cycles.
2. Rectangles indicate the groups.
3. Occording to Theorem 4, since there are no cycles, the minimal window size
   = max[all operations on A, all operations on B, all operations on C]
   = max[t(u)+t(w)+t(z), t(v)+t(x), t(y)]
   = max[6,2,3] = 6.




a.

| | | | | | |
|---|---|---|---|---|---|
| A | $u_i$ | $u_i$ | | | |
| B | | | $v_i$ | | |
| C | | | | | |

b.

| | | | | | | |
|---|---|---|---|---|---|---|
| A | $u_i$ | $u_i$ | $w_i$ | $w_i$ | | |
| B | | | $v_i$ | | $x_i$ | |
| C | $y_{i-1}$ | $y_{i-1}$ | | | | $y_i$ |

d.

| | | | | | |
|---|---|---|---|---|---|
| A | $u_i$ | $u_i$ | $w_i$ | $w_i$ | |
| B | | | $v_i$ | | $x_i$ |
| C | | | | | |

c.

| | | | | | | |
|---|---|---|---|---|---|---|
| A | $u_i$ | $u_i$ | $w_i$ | $w_i$ | $z_{i-1}$ | $z_{i-1}$ |
| B | | | $v_i$ | | $x_i$ | |
| C | | $y_{i-1}$ | $y_{i-1}$ | $y_{i-1}$ | | |

e.

Figure 12. Example 1

4. The groups are ordered I,II, & III.

5. Figure b shows the first group I positioned at the left window boundary.

6. In Figure c group II is added following I as closely as possible.

7. Since the predecessors of group III are groups I and II, column 6 is the first column which can be occupied by group III. Figure d shows an attempt to schedule group III starting in column 6.

8. Node y begins in column 6, is wrapped around and continued in columns 1 and 2 with a lower subscript.

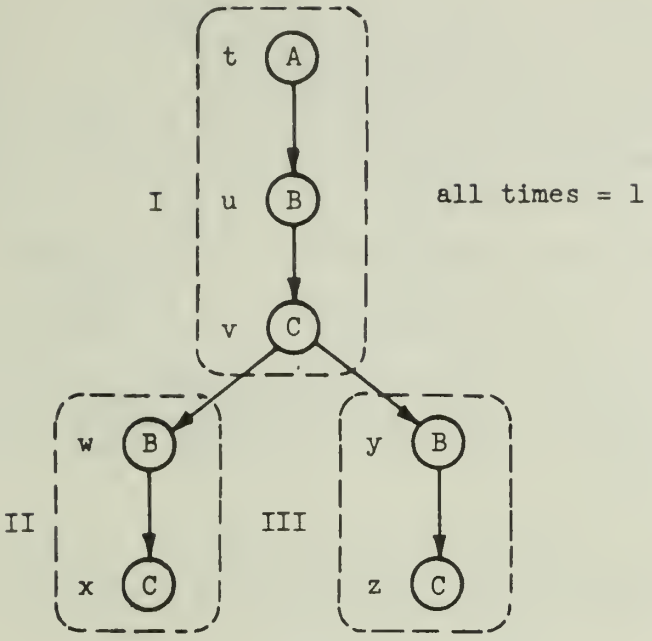
9. A conflict exists between node z and the already scheduled node w, since both nodes use resource A. Note that the link between nodes y and z is not a stretchable one.

10. In Figure e group III is moved to the right to begin in column 2 where it is evenually scheduled with no conflicts.

Although this is a simple example, it demonstrates the basic idea of the algorithm. In this case it was not necessary to increase the size of the window. Also, this is the best possible solution since the schedule is bound by resource A.

For the next example consider the graph shown in Figure 13.

1. The graph contains no dependency cycles.




a.

| | | | |
|---|---|---|---|
| A | $t_i$ | | |
| B | $w_{i-1}$ | $u_i$ | |
| C | | $x_{i-1}$ | $v_i$ |

b.

| | | | |
|---|---|---|---|
| A | $t_i$ | | |
| B | $w_{i-1}$ | $u_i$ | $y_{i-1}$ |
| C | | $x_{i-1}$ | $v_i$ |

c.

| | | | | |
|---|---|---|---|---|
| A | $t_i$ | | | $z_{i-1}$ |
| B | $w_{i-1}$ | $u_i$ | $y_{i-1}$ | |
| C | | $x_{i-1}$ | $v_i$ | |

d.

Figure 13. Example 2

2. Rectangles indicate the groups.

3. The initial window size
   = the minimal window size
   = max[all op.s on A, all op.s on B, all op.s on C]
   = max[t(t), t(u)+t(w)+t(y), t(v)+t(x)+t(z)]
   = max[1,3,3] = 3.

4. The groups are ordered I, II, and III.

5. Figure b shows groups I and II scheduled.

6. In Figure c an attempt is made to schedule group III by scheduling node y in column 3, but this results in a conflict between node z and node t in column 1.

7. After failing to fit group III in the other two columns, the window size is increased to 4 and the procedure is started from the beginning.

8. Figure d shows the final arrangement. Note that if the link between nodes y and z were stretchable, then this could have been scheduled in three instead of four columns.

For the last example refer to Figure 14.

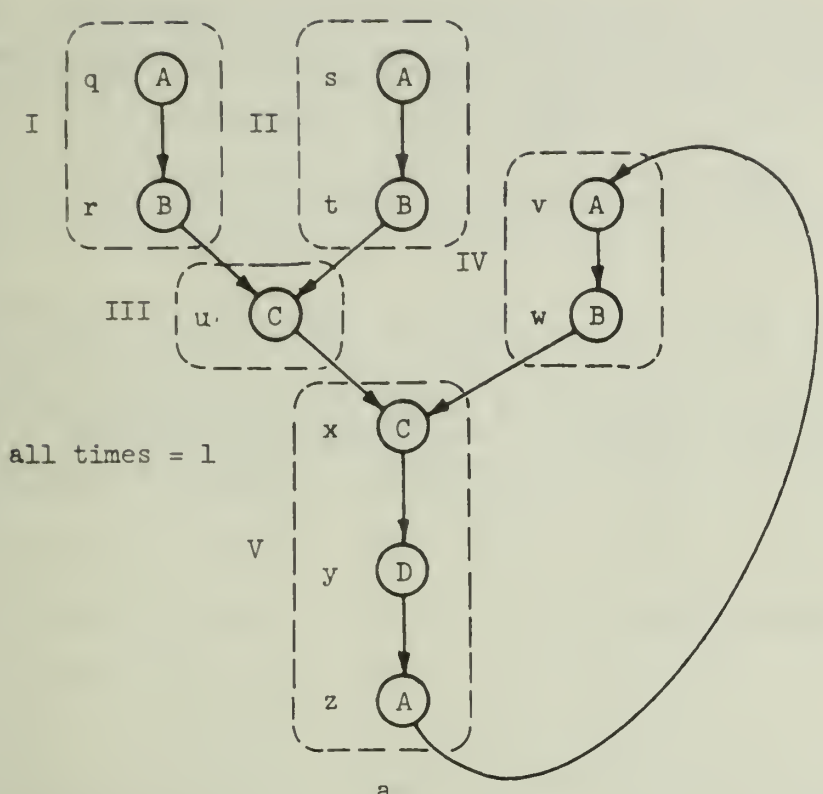
1. There exists a dependency cycle including the nodes v,w,x,y, and z.

2. Rectangles indicate the groups.

3. The initial window size
   = the minimal window size
   = max[length of the loop, all the op.s on A, all the op.s on B, all the op.s on C, all the op.s on D]
   = max[t(v)+t(w)+t(x)+t(y)+t(z),
   t(q)+t(s)+t(v)+t(z), t(r)+t(t)+t(w), t(u)+t(x),
   t(y)]
   = max[5,4,3,2,1] = 5.

4. The groups are ordered I,II,III,IV, and V.

5. Figure b shows groups I and II scheduled.




a.

| | | | | |
|---|---|---|---|---|
| A | $q_i$ | $s_i$ | | |
| B | | $r_i$ | $t_i$ | |
| C | | | | |
| D | | | | |

b.

| | | | | |
|---|---|---|---|---|
| A | $q_i$ | $s_i$ | $v_i$ | $z_{i-1}$ |
| B | | $r_i$ | $t_i$ | $w_i$ |
| C | | $x_{i-1}$ | | $u_i$ |
| D | | | $y_{i-1}$ | |

c.

| | | | | |
|---|---|---|---|---|
| A | $q_i$ | $s_i$ | $z_{i-1}$ | $v_i$ | |
| B | | $r_i$ | $t_i$ | | $w_i$ |
| C | $x_{i-1}$ | | | $u_i$ | |
| D | | $y_{i-1}$ | | | |

d.

Figure 14. Example 3

6. In Figure c groups III and IV are scheduled. Group V is scheduled beginning in column 2 after attempts to schedule it in columns 5 and 1 failed due to conflicts with nodes 1 and 3 respectively.

7. Upon observation it is found that Theorem 3 is violated. The occurrences of nodes v and y, which belong to the same cycle, are overlapped.

8. Group III is the one which contains the occurrences which overlap with those of the present group. Therefore we go back to the state of the window before the scheduling of group III (Figure b) and reschedule group III one column to the right in column 4. The final schedule is shown in Figure 14d.

D. Analysis of Algorithm.

In this section we analyze the algorithm in terms of an optimal solution. To develop the optimal solution we again make the assumption that registers are present in the machine when data is fanned-in or fanned-out in the graph. Later it will be shown that, in general, better results can be attained by adding more registers.

For the sake of simplicity in this argument, we will not consider the assignment of subscripts to ocurrences as was done in the algorithm presented in Section C. This is of no loss, since after the occurrences have all been scheduled, it is possible to assign subscripts subject to the condition of Theorem 2. The fact that the algorithm of Section C assigned subscripts in the process of scheduling

the groups does not affect its ability to generate a window nor its complexity. However, it does have a desirable effect which will be discussed later. Also we will assume that the graphs of our blocks contain no dependency cycles.

Recall that the use of registers enables us to divide the graph of a block into a number of nodes connected by unstretchable links. Each shape implies a fixed shape on the schedule chart. Our task is to fit the shapes together in a pattern (window) such that the period of repetition is minimal. One way to attack this problem is to assume a window size then try each shape in every column of the window. After every arrangement of the shapes is tried and none is found which enables all of the shapes to fit in the window, the window is increased in size and the shapes are tried again. Eventually a window size will be reached for which all the shapes fit. This solution implies a tree structure as in Figure 15. The number of levels in the tree is n, for n shapes; the fan-out at each node is w, where w is the number of columns in the present window. At each level, that is for each shape, we assign a column of the window for the beginning of a shape. The tree represents all possible assignments. As can be seen, it is possible to start several shapes in the same column. A depth first

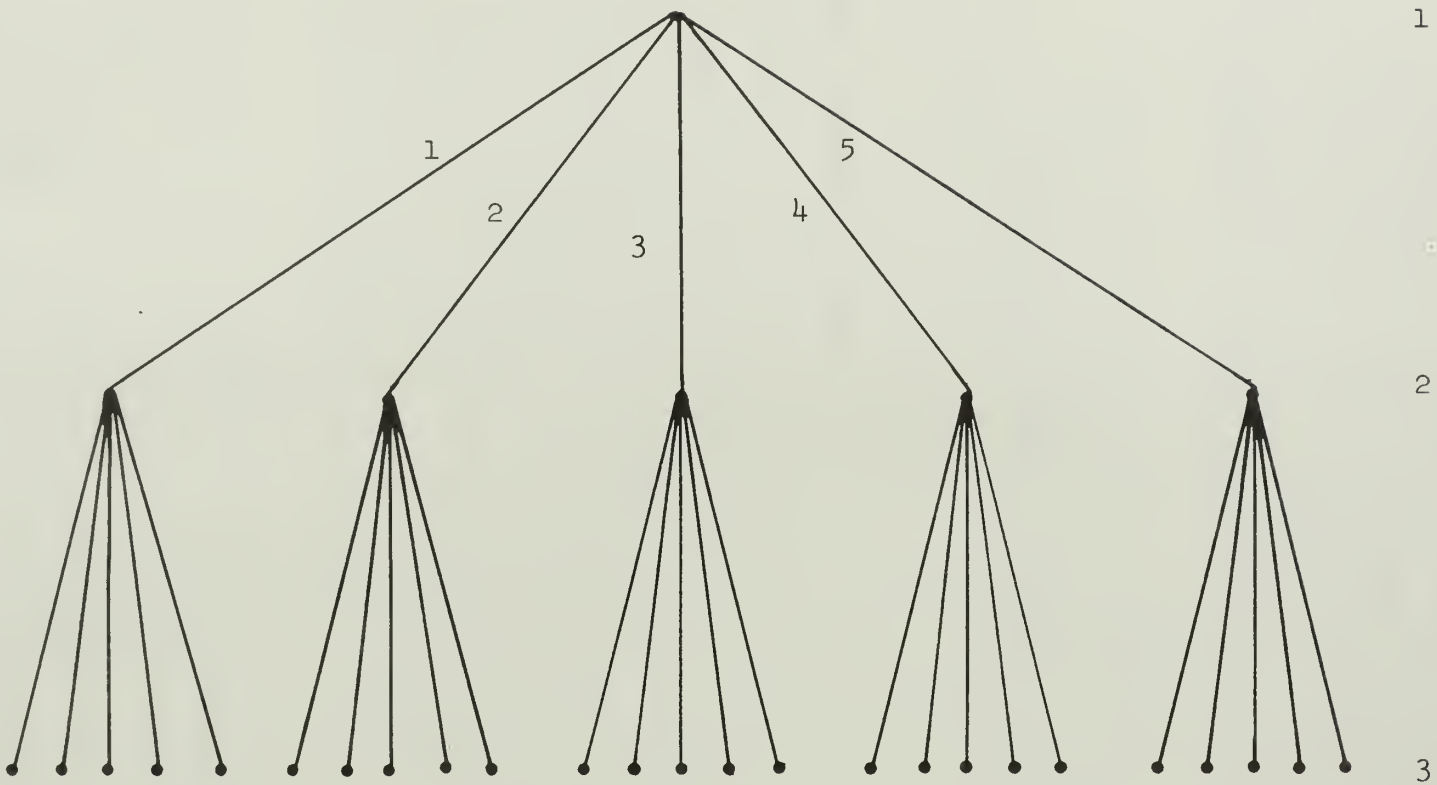



Figure 15. Structure of Optimal Solution for n = 3 and w = 5

traversal of the tree will guarantee the best possible arrangement of the shapes in the window. Since the window is repeated over and over, as before, it is possible to wrap-around part of a shape. If it is impossible to arrange the shapes in the window of present size then we increase the window size and try again. This yields a tree as before, now with the fan-out at each node increased by one.

Now if we assume the initial window size, $w_o$, to be the minimum as defined by Theorem 4, then the bound on the total number of combinations necessary to try before the optimal solution if found is:

$$(w_o)^{n-1} + (w_o + 1)^{n-1} + (w_o + 2)^{n-1} + \ldots$$

Here the series is bound by the maximum window size, which is the length of the window when all of the shapes are placed end to end with no attempt to overlap them. Because the window is repeated, the first shape can be assigned arbitrarily. This accounts for the n-1 rather than n in the superscripts.

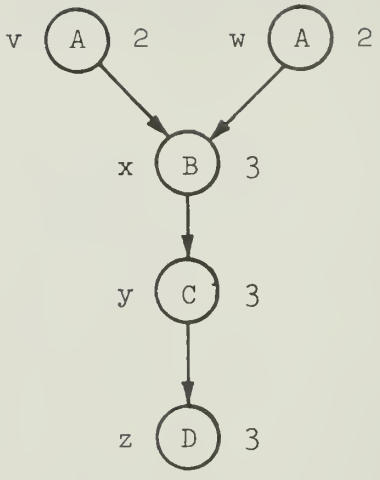
The algorithm of Section B is an improvement over this exhaustive search in that it does not try all possible combinations of the shapes. In terms of the tree structure, only one path from the top to the bottom is tried. At each node we try each successor one level down until the shape is fit in the window, then move down ignoring all other possible arrangements at that level. The result of this approach is that the number of combinations tried is bound by a polynomial in n rather than an by an exponential. Of course, the penalty is that there exists a possiblilty that the window size will have to be increased beyond its optimum. The results presented in Chapter V show that it is rarely the case in practice that the window size needs to be increased.

## E. HEAD and TAIL Generation

In this section we are concerned with the generation of the HEAD and TAIL sections of the chart. Obviously, it is possible to generate the HEAD and TAIL directly from the graph as was done with the BODY; but a simplier approach is to generate the HEAD and TAIL from the BODY. For this we

assume that we have the scheduling of one window of the BODY, including subscripts.

The HEAD can be considered as a number of BODY sections with the appropriate occurrences removed. The number of BODY sections is equal to one less than the number of iterations represented by the occurrences in one window. We will assume that the i-th iteration of the graph begins in the i-th window. Therefore the first HEAD section contains only those occurrences whose subscripts are i in the BODY window. The remaining sections each contain every occurrence that the last section contained along with some additional occurrences. The second section of the HEAD contains those occurrences from the BODY window whose subscripts are i and i-1. The third contains those subscripts that are i, i-1, and i-2, and so on for the remainder of the sections. Consider Figure 16 as an example. Figure a shows a graph and a window from its corresponding schedule chart. We observe that the window contains nodes from iterations i-3 through i. Therefore the HEAD contains three sections. Each section contains the appropriate occurrences from the window. The first section contains those occurrences with subscript equal to i. To these in the HEAD we assign the subscript of 1, in Figure b. The second section contains those occurrences from the window with subscripts $\geq$ i-1. Finally the last section




a.

| BODY | | | | | |
|---|---|---|---|---|---|
| | A | $v_i$ | $v_i$ | $w_i$ | $w_i$ |
| | B | $x_{i-1}$ | $x_{i-1}$ | $x_{i-1}$ | |
| | C | $y_{i-2}$ | $y_{i-2}$ | | $y_{i-1}$ |
| | D | $z_{i-3}$ | | $z_{i-2}$ | $z_{i-2}$ |

b.

| HEAD | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | $v_1$ | $v_1$ | $w_1$ | $w_1$ | $v_2$ | $v_2$ | $w_2$ | $w_2$ | $v_3$ | $v_3$ | $w_3$ | $w_3$ |
| | B | | | | | $x_1$ | $x_1$ | $x_1$ | | $x_2$ | $x_2$ | $x_2$ | |
| | C | | | | | | | | $y_1$ | $y_1$ | $y_1$ | | $y_2$ |
| | D | | | | | | | | | | | $z_1$ | $z_1$ |

c.

| TAIL | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | | | | | | | | |
| | B | $x_9$ | $x_9$ | $x_9$ | | | | | |
| | C | $y_8$ | $y_8$ | | $y_9$ | $y_9$ | $y_9$ | | |
| | D | $z_7$ | | $z_8$ | $z_8$ | | | $z_9$ | $z_9$ | $z_9$ |

Figure 16. HEAD and TAIL Generation

contains those occurrences from the window with subscripts $\geq$ i-2.

Generation of the TAIL is very similar to that of the HEAD. The TAIL contains the same number of sections of the HEAD. In general the window contains occurrences from iterations i-d through i of the graph.

The first section of the TAIL contains all the occurrences from the window except those having the subscript i. The second section contains all the occurrences except those having the subscripts i and i-1, and so on for all sections. As a result, the last section of the TAIL contains only those occurrences with the subscript i-d. Figure 16c demonstrates this. Here we see that the first section of the TAIL contains all the occurrences from the window with subscripts $\leq$ i-1. The second section contains those with subscripts $\leq$ i-2. The last section contains only those occurrences from the window with subscript i-3.

## F. Joining Blocks

Up to this point we have been dealing only with blocks of nodes, where each block is executed iteratively and itself contains no iteration loops. Most algorithms of interest are composed of these blocks which may be nested several levels deep in iteration loops (recall Figure 4). If we wish to schedule an entire algorithm we must first schedule each block then combine the results. A good deal of effort was exerted to provide for the generation of the most efficient schedule chart for each block. And rightly so, because most savings in time and efficiency come from a tight body section for each block. In addition, the manner in which the schedules of each block are combined has an effect on the total result. That is, if we can overlap the scheduling of two adjacent blocks, then the chart for the entire algorithm will be reduced in total length. Furthermore, if the two blocks are in an iteration loop then the total length of the algorithm will be reduced by the amount of the overlap times the total number of iterations of the loop. We now turn our attention to the problem of connecting the schedule charts of two adjacent blocks. Since we are able to schedule all of the blocks, if we can connect any two blocks, then with the proper control we can schedule the entire algorithm. The next chapter is concerned with scheduling the entire algorithm.

When considering the joining of schedule charts several possiblities exist, each with its own advantages. We will present three of these possibilities by way of an example.

Figure 17a shows the end of one schedule chart and the beginning of another. The simplest way two join two charts is by butting them together with no attempt to overlap the two. The result of this is shown in Figure b. A slightly more complicated scheme is shown in Figure c. Here the two charts were slid together until they met. In this case it was possible to overlap the two charts by one column. A larger overlap can be acheived by moving the charts even closer together. To do this we hop one chart over the other to a spot at which it fits. The result of this shown in Figure d.

Since two methods involved an overlap of charts it is necessary to be concerned with dependencies between charts. We must be careful not to violate a data dependence. This is one advantage of the first method. For the other two methods the greatest amount of overlap is limited by the data dependencies between blocks.

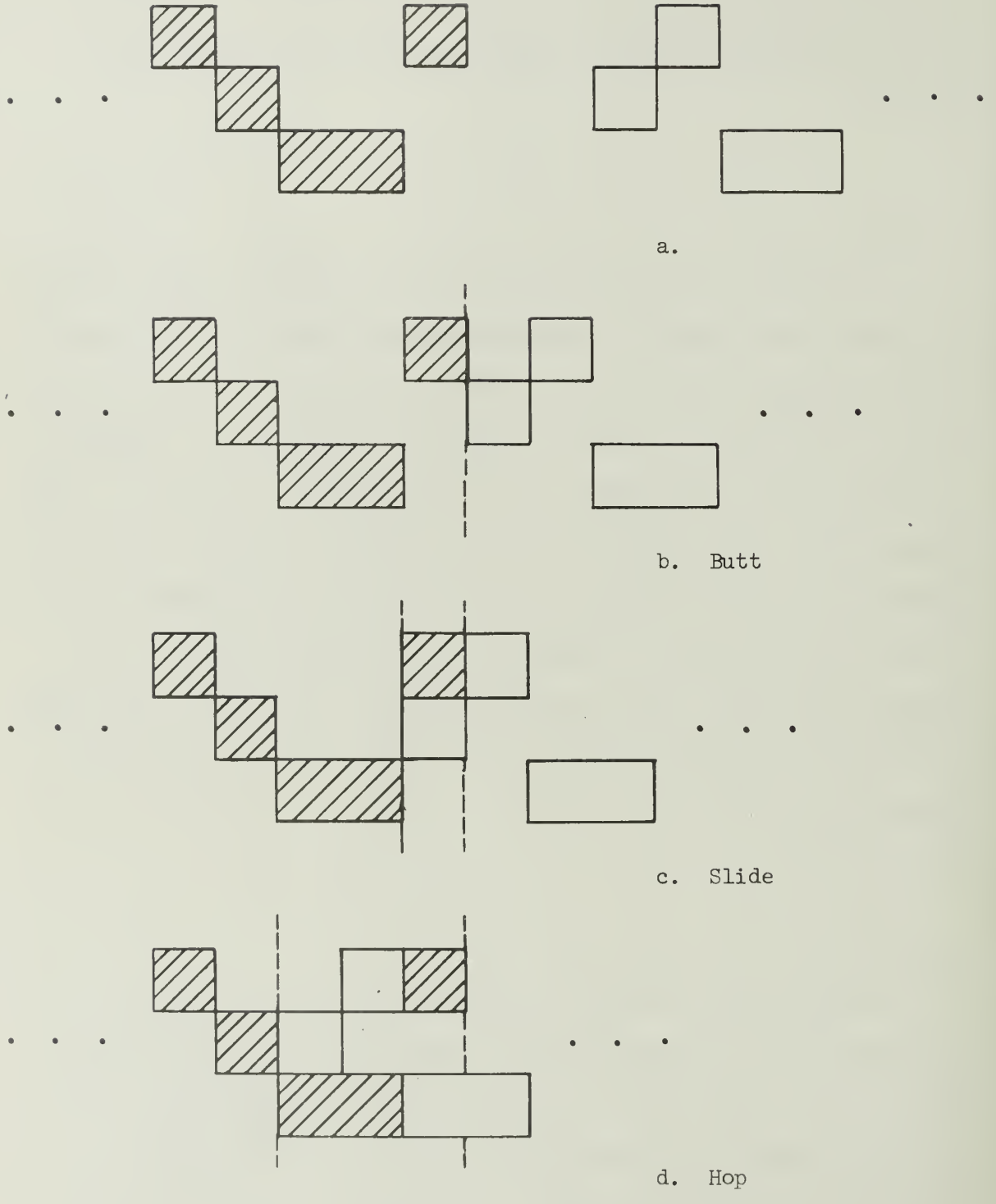



Figure 17. Joining Blocks

Another possibility exists for joining two adjacent blocks. This approach does not assume that we already have the scheduling of the TAIL and HEAD sections before we begin. The idea is to simultaneously generate the TAIL of the first block and the HEAD of the next directly from their respective graphs. In this manner we can achieve the shortest total length for the chart. The disadvantage of this approach is that the HEAD and TAIL sections must be regenerated each time a block is considered. Whereas with the other three methods they can be generated once and then reused each time that block occurs.

## G. Some Comments on Hardware

When considering the various methods for joining schedule charts one basic tradeoff is always present; the greatest amount of overlap between adjacent blocks, which implies the shortest overall chart length, is paid for by a more complex scheduling and joining algorithm. If the scheduling is considered as microprogram control, this means that it is possible to acheive a fast execution time at the cost of a complex microprogram control unit.

Let us assume that we have scheduled a collection of blocks and wish to store the results as part of a control store memory. The schedule charts can either be stored as a series of microinstructions, where each block corresponds to a machine language instruction, or at a lower level such as nanoprogram store. If the latter is the case then the structure of the computation algorithm, including branches and iteration loops enclosing collections of blocks, can be stored at the microprogram level. For the former, the task of properly sequencing the blocks is the responsibilty of the compiler or the programmer. Since the BODY of the chart is periodic, all we need to store is one window. To produce the correct control signal we simply cycle through the window an appropriate number of times. To generate the control signals throughout the HEAD and TAIL portions of the chart the window is cycled with the appropriate signals ignored. As was implied earlier this requires the the subscript information be stored along with the scheduling of the nodes, in the control memory. Also a mechanism must be present which can examine the subscripts to determine which signals should be ignored and which should be passed on. A far more simple approach, but one which requires more storage, is to store the predermined HEAD and TAIL portions of the graph along with one window.

The next question to answer is that of the method of joining blocks to produce the correct control signals throughout the the exectution of the algorithm. Surely the simplest way to do this is to use the butt method described previously. Here, when the TAIL of one block is completed, the HEAD of the next is started, without overlapping the two. Although this method does not yield the best possible execution time, it is by far the simplest to implement. To overlap the execution of adjacent blocks implies the need to simultaneously examine the end of the chart of the first with the beginning of the chart of the second. This cannot be done dynamically and therefore must be predeterminded. Also, if the blocks are to correspond to machine instructions, then the ability must exist for the specification of a begin time for each instruction relative to the begin time of the last. Alternately, if the sequencing of the blocks is to be performed through the use of a microprogram making calls to the blocks stored in a nanoinstruction store, the microprogram needs to contain this proper timing information.

# IV. FAPGEN - A HIGH LEVEL DESCRIPTION

At this point we have all the tools necessary to schedule the execution of an entire algorithm. Now the problem becomes one of expressing the structure of the algorithm and then dealing with it. For this we turn our attention to a PL/1 program which was developed for this purpose. The program is called FAPGEN for FAPAS chart generator, where FAPAS is an acronym for the row labels (fetch, align, process, align, and store) of schedule charts for the BSP. The ideas and methods of the last three chapters are incorporated into FAPGEN and all come into play at various times in the scheduling of an algorithm. In this manner FAPGEN serves as a tool for the simulation of an algorithm while providing a means to study the methods of simulation. The methods of simulation are important because they imply in a direct sense the machine design. For example, the blocks of an algorithm can correspond to machine level instructions on the target machine. If this is the case, then questions about HEAD and TAIL generation and the joining of blocks become real problems when designing the control unit for the target machine.

FAPGEN is arranged as a collection of user callable PL/1 procedures all communicating through common data structures. Each call to a procedure in FAPGEN initiates an action and a modification of the common data structures. By properly sequencing calls to FAPGEN, the structure of any algorithm can be expressed. The user also has the benefits of the standard PL/1 constructs. For example, through the use of the IF THEN ELSE construct certain sections of an algorithm can be dependent on some external parameters.

Details of the use of FAPGEN are included in the appendix. Here we will present a brief high level explanation of its structure and operation. To aid in the discussion we will use the algorithm of Figure 4. The FAPGEN constructs at our diposal are; DOLOOP for iteration loops within the algorithm definition, MAP which generates the schedule chart for each block, and ENDO for ending an iteration loop. Later we will add DEPEND to indicate data dependencies between nodes in different blocks. The algorithm can be defined as follows:

```
ALGORITHM: PROC(alpha,beta);
           DECLARE DOLOOP, MAP, ENDO EXTERNAL ENTRY;
```

```
CALL DOLOOP (N1) ;
     CALL MAP (Block1,N2) ;
     IF alpha < beta
        THEN CALL MAP (Block2,N3) ;
        ELSE CALL MAP (Block3,N4) ;
     CALL DOLOOP (N5) ;
          CALL MAP (Block4,N6) ;
          CALL MAP (Block5,N7) ;
          CALL ENDO ;
     CALL ENDO ;
END ALGORITHM;
```

A graph of each block is stored by the user in a library. So when FAPGEN is called to schedule a block, its graph can be looked up. In this manner blocks can be considered as user defineable primitives for algorithm definition. The important common data structures internal to FAPGEN for the interpretation of an algorithm definition are; a stack, a collection of records (one for each call to DOLOOP or MAP), and a structure in which to store the schedule charts. Each schedule chart can be considered to be two 2-dimensional arrays (one for node numbers and one for subscripts). We will now work through the example step by step to show the internal methods of FAPGEN. This explanation is theoretical; in practice some modifications

are present. For this example assume that procedure ALGORITHM was called with alpha = beta.

CALL DOLOOP(N1) creates a record indicating that this is the beginning of a loop which should be iterated n1 times. A pointer to this record is then pushed on the stack. Next is CALL MAP(Block1,N2). Procedure MAP looks up the graph structure for Block1 in the library of stored graphs. From this it generates a schedule chart using a procedure as was described earlier in Chapter III, Section C. A record is created which contains a pointer to the chart and an identification of this as being a schedule chart. A pointer to this record is pushed on the stack. Since alpha = beta, the next step is CALL MAP(Block3,N4). As before, a schedule chart defining the HEAD, BODY, and TAIL sections are generated, the proper record is generated, and a pointer to the record pushed on the stack. Execution continues in this manner until a CALL ENDO is encountered. Procedure ENDO first sets up storage for a temporary schedule chart, call this TEMP1. Evenually this chart will contain the scheduling of this entire DOLOOP. Next a pointer is popped from the stack. Since it points to a record indicating a schedule chart, the chart is copied to the temporary space (TEMP1). Again ENDO pops a pointer from

the stack. This points to the record of another schedule chart (Block4) which is joined in TEMP1 with what is already there (the chart of Block5). Methods for joining blocks were dicussed in Chapter III. The particular method used can be determined by a global option. Now TEMP1 can be considered to be in the same form as the schedule charts of the individual blocks. Its HEAD is the HEAD of Block4, its TAIL is the TAIL of Block5, and its BODY is composed of the BODY and TAIL of Block4 and the HEAD and BODY of Block5. Next, procedure ENDO pops a pointer and sees that this is the beginning of the loop. Since there are no more blocks within this DOLOOP, at this point TEMP1 represents the scheduling of one iteration of this DOLOOP. ENDO then joins the appropriate number of charts (in this case N5) using TEMP1 as on iteration. The result is stored in TEMP1 and a record pointing to it is generated. Then a pointer to this record is pushed on the stack and control is returned to ALGORITHM. Next is another call to ENDO. Here TEMP2 is set up to store the temporary schedule chart. As before pointers are popped one at a time from the stack. The charts of each previously scheduled chart, including blocks and temporary charts, are combined in TEMP2, each one being joined ahead of the last. ENDO continues popping pointers and joining charts until one pointer indicates that this is the beginnıng of the DOLOOP. Then n1 copies of the chart in TEMP2 are joined into one.

In practice this would require extensive storage for the charts. Economy of storage can be achieved by abbreviating the BODY sections.

This is the minimum amount of work needed to schedule an algorithm. In practice it may be desirable to have some additional capabilities. One such capability is the ability to express, in the algorithm definition, data dependencies between nodes in different blocks. For this we need to add an internal structure to FAPGEN to hold the dependence information. Along with this is another entry point to FAPGEN to fill in the dependence information and a mechanism in ENDO to deal with the data dependencies. For example suppose that the operation indicated by node 1 in Block5 is dependent on the result of the operation of node 5 in Block4. This can be incorporated into procedure ALGORITHM as follows:

```
CALL MAP(Block4,N6) ;
CALL DEPEND(node 1,Block5,node 5,Block4) ;
CALL MAP (Block5,N7) ;
```

The call to procedure DEPEND creates a structure to record the dependence. Later when ENDO is combining the charts of Block4 and Block5 it refers to this dependence

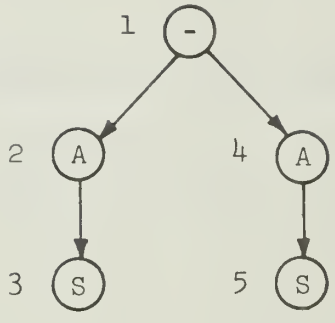
structure. The dependence structure indicates that the occurrence of node 1 in Block5 cannot be overlapped with the final occurrence of node 5 in Block4.

Other options can be added to this basic scheme. These include; the mechanisms and data structures for the tallying of the resource times, collection of statistics on frequencies of use of the blocks in the library, use of sneak paths, and various print options.

# V. SOME RESULTS, EXTENSIONS, and PROBLEMS.

In this chapter we futher investigate the proposed algorithm for the scheduling of blocks. We discuss a possible extension to our approach and some potential problems.

Figures 18 through 23 show the results of scheduling some blocks of nodes using the procedure of Chapter III, Section C. The blocks are representative of a set which has been compiled after the examination and definition of several algorithms for the solution of linear recurrence relations. The machine is an array processor of the type of the BSP. Here the symbols inside the nodes correspond to some operation performed by each resource of the machine, the structure of which is shown in Figure 2. The processors are designed in such a way that they all execute the same operation simultaneously. We assume that each block is iterated a number of times much greater than one. Figures 18a through 23a show the scheduling of the HEAD, a representative BODY section, and the TAIL section of the charts for the case where memory time = alignment time = 1, add time = subtract time = multiplication time = 2, division




BODY

| | | | |
|---|---|---|---|
| M | $3_7$ | $5_8$ | |
| A | | | |
| P | $1_9$ | $1_9$ | |
| A | $4_8$ | $2_8$ | |

HEAD

| | | | | |
|---|---|---|---|---|
| M | | | | $5_7$ |
| A | | | | |
| M | $1_7$ | $1_7$ | $1_8$ | $1_8$ |
| A | | | $4_7$ | $2_7$ |

TAIL

| | | | | |
|---|---|---|---|---|
| M | $3_8$ | $5_9$ | $3_9$ | |
| A | | | | |
| P | | | | |
| A | $4_9$ | $2_9$ | | |

a.

BODY

| | | | | |
|---|---|---|---|---|
| M | $3_7$ | | $5_8$ | |
| A | | | | |
| P | $1_9$ | $1_9$ | $1_9$ | $1_9$ |
| A | $4_8$ | $4_8$ | $2_8$ | $2_8$ |

HEAD

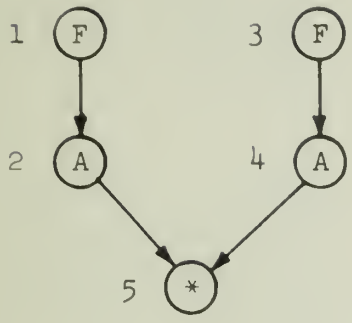
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| M | | | | | | | $5_7$ | |
| A | | | | | | | | |
| P | $1_7$ | $1_7$ | $1_7$ | $1_7$ | $1_8$ | $1_8$ | $1_8$ | $1_8$ |
| A | | | | | $4_7$ | $4_7$ | $2_7$ | $2_7$ |

TAIL

| | | | | | |
|---|---|---|---|---|---|
| M | $3_8$ | | $5_9$ | | $3_9$ |
| A | | | | | |
| P | | | | | |
| A | $4_9$ | $4_9$ | $2_9$ | $2_9$ | |

b.

Figure 18. Scheduling a Block




a.

BODY

| | | | | | | |
|---|---|---|---|---|---|---|
| M | $1_9$ | $3_9$ | | | | |
| A | $4_8$ | $2_9$ | | | | |
| P | $5_6$ | $5_7$ | | | | |
| A | | | | | | |

HEAD

| | | | | | | |
|---|---|---|---|---|---|---|
| M | $1_6$ | $3_6$ | $1_7$ | $3_7$ | $1_8$ | $3_8$ |
| A | | $2_6$ | $4_6$ | $2_7$ | $4_7$ | $2_8$ |
| P | | | | | | $5_6$ |
| A | | | | | | |

TAIL

| | | | | | |
|---|---|---|---|---|---|
| M | | | | | |
| A | $4_9$ | | | | |
| P | $5_7$ | $5_8$ | $5_8$ | $5_9$ | $5_9$ |
| A | | | | | |

b.

BODY

| | | | | | | |
|---|---|---|---|---|---|---|
| M | $1_9$ | | $3_9$ | | | |
| A | | $2_9$ | $2_9$ | $4_9$ | $4_9$ | |
| P | $5_8$ | $5_8$ | $5_8$ | $5_8$ | $5_8$ | $5_9$ |
| A | | | | | | |

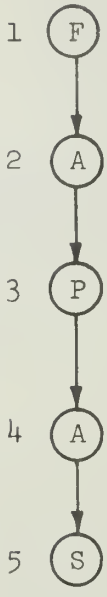
HEAD

| | | | | | | |
|---|---|---|---|---|---|---|
| M | $1_8$ | | $3_8$ | | | |
| A | | $2_8$ | $2_8$ | $4_8$ | $4_8$ | |
| P | | | | | | $5_8$ |
| A | | | | | | |

TAIL

| | | | | | |
|---|---|---|---|---|---|
| M | | | | | |
| A | | | | | |
| P | $5_9$ | $5_9$ | $5_9$ | $5_9$ | $5_9$ |
| A | | | | | |

Figure 19. Scheduling a Block




BODY

| | | |
|---|---|---|
| M | $1_9$ | $5_7$ |
| A | | $2_9$ |
| P | $3_8$ | $3_8$ |
| A | $4_7$ | |

HEAD

| | | | | |
|---|---|---|---|---|
| M | $1_7$ | | $1_8$ | |
| A | | $2_7$ | | $2_8$ |
| P | | | $3_7$ | $3_7$ |
| A | | | | |

TAIL

| | | | | |
|---|---|---|---|---|
| M | | $5_8$ | | $5_9$ |
| A | | | | |
| P | $3_9$ | $3_9$ | | |
| A | $4_8$ | | $4_9$ | |

a.

BODY

| | | |
|---|---|---|
| M | $1_9$ | $5_6$ |
| A | $2_8$ | $2_9$ |
| P | $3_7$ | $3_8$ |
| A | $4_6$ | $4_7$ |

HEAD

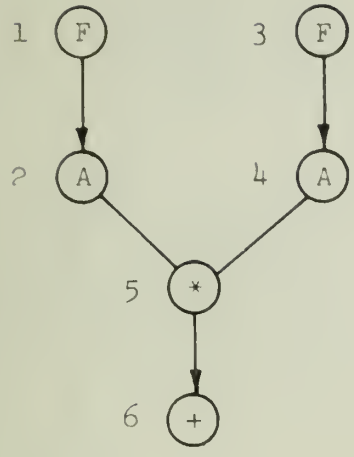
| | | | | | | |
|---|---|---|---|---|---|---|
| M | $1_6$ | | $1_7$ | | $1_8$ | |
| A | | $2_6$ | $2_6$ | $2_7$ | $2_7$ | $2_8$ |
| P | | | | $3_6$ | $3_6$ | $3_7$ |
| A | | | | | | $4_6$ |

TAIL

| | | | | | | |
|---|---|---|---|---|---|---|
| M | | $5_7$ | | $5_8$ | | $5_9$ |
| A | $2_9$ | | | | | |
| P | $3_8$ | $3_9$ | $3_9$ | | | |
| A | $4_7$ | $4_8$ | $4_8$ | $4_9$ | $4_9$ | |

b.

Figure 20 Scheduling a Block




a.

BODY

| | | | | |
|---|---|---|---|---|
| M | $1_9$ | $3_9$ | | |
| A | | $2_9$ | $4_9$ | |
| P | $5_8$ | $6_8$ | $6_8$ | $5_9$ |
| A | | | | |

HEAD

| | | | | |
|---|---|---|---|---|
| M | $1_8$ | $3_8$ | | |
| A | | $2_8$ | $4_8$ | |
| P | | | | $5_8$ |
| A | | | | |

TAIL

| | | | | |
|---|---|---|---|---|
| M | | | | |
| A | | | | |
| P | $5_9$ | $6_9$ | $6_9$ | |
| A | | | | |

b.

BODY

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| M | $1_9$ | | $3_9$ | | | | | | | |
| A | | $2_9$ | $2_9$ | $4_9$ | $4_9$ | | | | | |
| P | $5_8$ | $6_8$ | $6_8$ | $6_8$ | $6_8$ | $5_9$ | $5_9$ | $5_9$ | $5_9$ | $5_9$ |
| A | | | | | | | | | | |

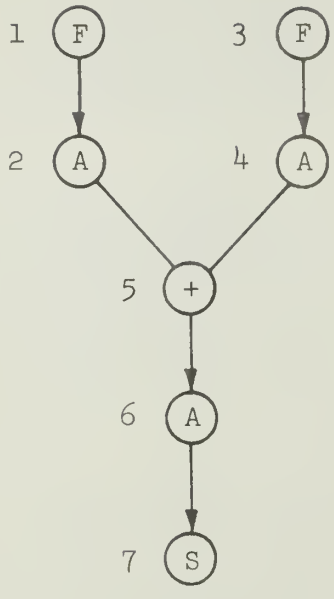
HEAD

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| M | $1_8$ | | $3_8$ | | | | | | | |
| A | | $2_8$ | $2_8$ | $4_8$ | $4_8$ | | | | | |
| P | | | | | | $5_8$ | $5_8$ | $5_8$ | $5_8$ | $5_8$ |
| A | | | | | | | | | | |

TAIL

| | | | | | |
|---|---|---|---|---|---|
| M | | | | | |
| A | | | | | |
| P | $5_9$ | $6_9$ | $6_9$ | $6_9$ | $6_9$ |
| A | | | | | |

Figure 21. Scheduling a Block




a.

BODY

| M | | | |
|---|---|---|---|
| A | | | |
| P | | | |
| A | | | |

HEAD

| M | | | | | | |
|---|---|---|---|---|---|---|
| A | | | | | | |
| P | | | | | | |
| A | | | | | | |

TAIL

| M | | | | | | |
|---|---|---|---|---|---|---|
| A | | | | | | |
| P | | | | | | |
| A | | | | | | |

b.

BODY

| M | | | | |
|---|---|---|---|---|
| A | | | | |
| P | | | | |
| A | | | | |

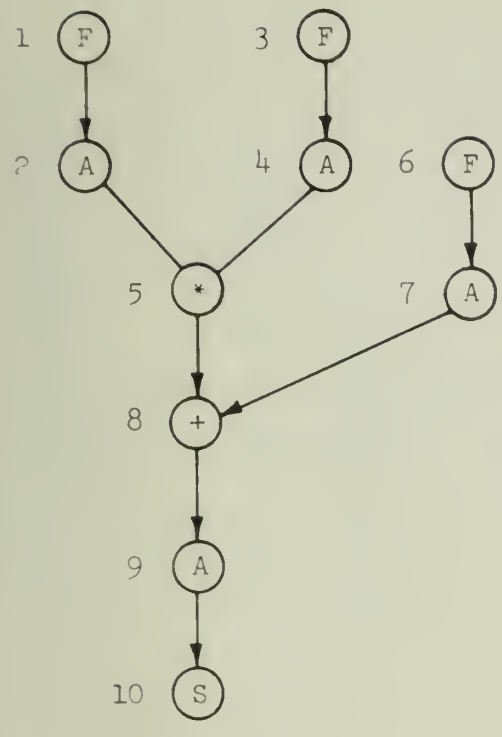
HEAD

| M | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | | | | | | | | |
| P | | | | | | | | |
| A | | | | | | | | |

TAIL

| M | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | | | | | | | | |
| P | | | | | | | | |
| A | | | | | | | | |

Figure 22. Scheduling a Block




a.

BODY

| | | | | | |
|---|---|---|---|---|---|
| M | $1_9$ | $3_9$ | $6_9$ | $10_8$ | |
| A | | $2_9$ | $4_9$ | $7_9$ | |
| P | $8_8$ | $8_8$ | | $5_9$ | $5_9$ |
| A | | | $9_8$ | | |

HEAD

| | | | | | |
|---|---|---|---|---|---|
| M | $1_8$ | $3_8$ | $6_8$ | | |
| A | | $2_8$ | $4_8$ | $7_8$ | |
| P | | | | $5_8$ | $5_8$ |
| A | | | | | |

TAIL

| | | | | |
|---|---|---|---|---|
| M | | | | $10_9$ |
| A | | | | |
| P | $8_9$ | $8_9$ | | |
| A | | | $9_9$ | |

b.

BODY

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| M | $1_9$ | | $3_9$ | | $6_9$ | | | $10_8$ | | |
| A | | $2_9$ | $2_9$ | $4_9$ | $4_9$ | $7_9$ | $7_9$ | | | |
| P | $5_8$ | $8_8$ | $8_8$ | $8_8$ | $8_8$ | $5_9$ | $5_9$ | $5_9$ | $5_9$ | $5_9$ |
| A | | | | | | $9_8$ | $9_8$ | | | |

HEAD

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| M | $1_8$ | | $3_8$ | | $6_8$ | | | | | |
| A | | $2_8$ | $2_8$ | $4_8$ | $4_8$ | $7_8$ | $7_8$ | | | |
| P | | | | | | $5_8$ | $5_8$ | $5_8$ | $5_8$ | $5_8$ |
| A | | | | | | | | | | |

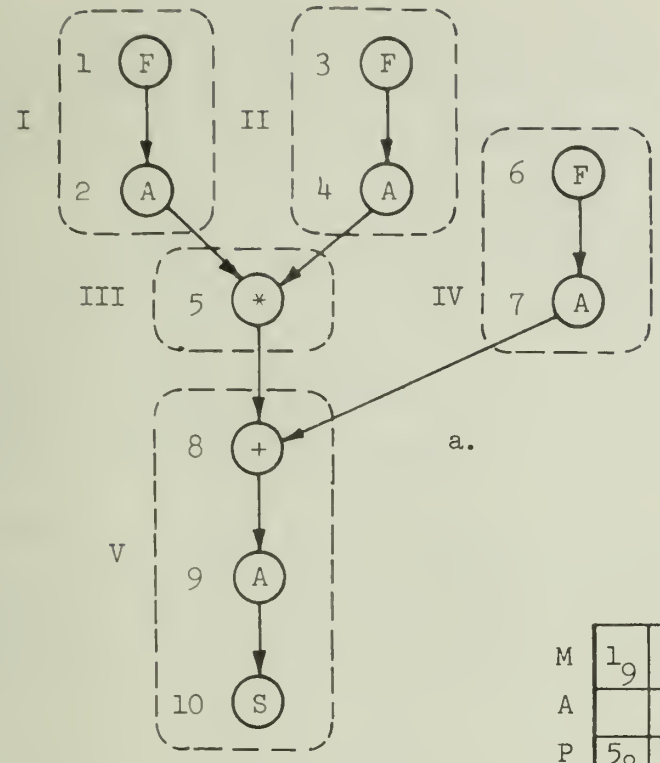
TAIL

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| M | | | | | | | | $10_9$ |
| A | | | | | | | | |
| P | $5_9$ | $8_9$ | $8_9$ | $8_9$ | $8_9$ | | | |
| A | | | | | | $9_9$ | $9_9$ | |

Figure 23. Scheduling a Block

time = 3, and null operation processor time = 2. All times may be considered to be in terms of machine cycles, but only their relative values are important. Memory operations include fetch (F) and store (S). The method of Chapter III Section C was employed for HEAD and TAIL generation. Figures 18b through 23b show the scheduling of the same blocks for the case where memory time = 1, alignment times = 2, add time = subtract time = 4, multiplication time = 6, division time = 8, and null operation processor time = 2.

In all cases except Figure 23a we were able to schedule the nodes of the blocks in a window of minimum size as defined by theorem 4. Let us examine this case to see what happened to force an increase in the window size. Figure 24a shows the graph of Figure 23a divided up into groups of nodes connected by unstretchable links, and the ordering of the groups for scheduling. In Figure b we see the progress of the scheduling algorithm up to the point where the window size had to be increased. Groups I through IV had been scheduled along with nodes 8 and 9 of group I. But it was found impossible to schedule node 10 without stretching the link between nodes 9 and 10. Therefore the window size was increased to 5 which proved to be sufficient. The new result is shown in Figure c. In Figure




b.

| | | | | |
|---|---|---|---|---|
| M | $1_9$ | $3_9$ | $6_9$ | |
| A | | $2_9$ | $4_9$ | $7_9$ |
| P | $5_8$ | $8_8$ | $8_8$ | $5_9$ |
| A | | | | $9_8$ |

c.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| M | $1_8$ | $3_8$ | $6_8$ | $10_7$ | | $1_9$ | $3_9$ | $6_9$ | $10_8$ | |
| A | | $2_8$ | $4_8$ | $7_8$ | | | $2_9$ | $4_9$ | $7_9$ | |
| P | $8_7$ | $8_7$ | | $5_8$ | $5_8$ | $8_8$ | $8_8$ | | $5_9$ | $5_9$ |
| A | | | $9_7$ | | | | | $9_8$ | | |

d.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| M | $1_7$ | $3_7$ | $6_7$ | $10_5$ | $1_8$ | $3_8$ | $6_8$ | $10_6$ | $1_9$ | $3_9$ | $6_9$ | $10_7$ |
| A | | $2_7$ | $4_7$ | $7_7$ | | $2_8$ | $4_8$ | $7_8$ | | $2_9$ | $4_9$ | $7_9$ |
| P | $5_6$ | $8_6$ | $8_6$ | $5_7$ | $5_7$ | $8_7$ | $8_7$ | $5_8$ | $5_8$ | $8_8$ | $8_8$ | $5_9$ |
| A | | | $9_5$ | | | | $9_6$ | | | | $9_7$ | |

Figure 24. Effect of Adding a Stretchable Link

d we return to the original window size of 4 columns, but this time it is assumed that there exists a input register in the output alignment network. This implies that the link between nodes 8 and 9 is a stretchable one. As a result we are able to schedule the block in a window of 4 columns instead of 5. This result can be generalized and it becomes apparent that with a sufficient number of registers between resources in a machine, the minimum window size is always large enough for the scheduling of the block. This presents an interesting question. How many registers is a sufficient number?

In Figures c and d one iteration of the graph has been shaded to show another result of adding a register at the input of the output alignment network. After the register has been added, the iteration becomes spread out. Although this is a result of adding the register, this spreading out effect is characteristic of our approach in general and presents a problem. We will address this in the next section.

Until now we have assumed that the number of iterations of the group of nodes composing the blocks was >> 1. For iteration counts equal to and close to 1, the

procedure presented in Chapter III Section C may not produce the best results.

Consider first the case where the iteration count = 1. Here the concept of a window is not easily applicable, but the procedure can be adapted by defining a very large initial window size to prevent wrap-around.

For blocks with iteration counts > 1 it is possible to generate their schedule charts either of two ways. One method is to schedule one iteration of the graph and then connect the proper number of these charts together. Alternately, it is possible to use the procedure presented for scheduling blocks of nodes. Using this method we assume that some of the links within the graph are stretchable. As seen earlier this has the effect of increasing the length of each iteration. For this reason it is possible that the procedure may produce a schedule chart which is greater in length than one produced by simply joining together the proper number of charts for one iteration. This problem is most severe when the number of iterations represented in the window is sufficiently large with respect to the number of iterations in the graph. To compare the two methods let us find a formula for the total length of the schedule chart as produced by both methods.

For the first method, that is where we do not use wrap-around, if we let r be the length of one iteration, then the total length of the chart for n iterations is $\leq$ n*r. If we were to butt the iterations together, the total length would be exactly n*r; but since it may be possible to overlap iterations together the total length may be < n*r. For the procedure using wrap-around, if we let w be the size of the window and d be one less than the number of iterations represented in the window, then the total length of the chart for n iterations is = (n+d)*w. This is arrived at by recognizing that there are d sections each for the HEAD and the TAIL and n-d sections in the BODY. If we assume that each HEAD and TAIL section is the size of the window then we have [d+(n-d)+d]*w = (n+d)*w. Although this may seem to imply that d must always be $\leq$ n, this is not the case. If d > n the formula (n+d)*w gives the correct length of the chart but the distinction between the HEAD, BODY, and TAIL of the chart is no longer clear.

For an example of where the first method yields a better result assume that for a graph, one iteration can be scheduled in 7 columns. Therefore r = 7. Alternately, by using the procedure presented we schedule one window in 5 columns with d = 1 (one wrap-around). Therefore by the first method for two iterations of the graph the total length is $\leq$ 14. Using the second method we get a total

length of 15.

By equating the two formulas we can solve for a range of n where the first method yields the best result. The result is that for $n < d*[w/(1-w)]$, the first approach yields the shortest chart.

# VI. CONCLUSION

We have presented some methods and procedures for the scheduling of the execution of a class of algorithms on a pipelined architecture. Advantage was taken of the iterative nature of sections of the computations. With this we found that much of the low level information can be predetermined and stored as part of the control section of the machine.

When simulating the operation of a machine with this control structure, we found it necessary to develop a procedure to schedule the execution of one iterative section of an algorithm or one vector operation (see Chapter III Section C). One important result with respect to this is that for the cases of practical interest, the initial window size seldom needed to be increased. The significance of this is that a very good estimate of the size of the schedule chart can found with little work. This can be used, perhaps, to choose quickly one algorithm out of several for a particular application or to perform an approximate simulation.

In the case of control, it is desirable to find a relatively small set of vector type operations or blocks which can form a set of primitives in terms of which all other computations can be expressed [6]. With this being the case and since we would be working with a fixed set of resource times, there would be no need for the procedure presented for producing the schedule for each block. Here the luxury could be afforded of using the exhaustive search method to schedule this limited set of primitives, since each only need be done once. Therefore the procedure presented is only needed when there exists a very large set of unique blocks or when it is desirable to recompute the schedule charts for different sets of resource times.

# REFERENCES

[1] D. J. Kuck and R. Stokes,
"The Burroughs Scientific Processor (BSP),"
submitted for publication.

[2] M. J. Wolfe,
"Techniques for Improving the Inherent Parallelism in Programs,"
Master's Thesis, University of Illinois at Urbana, UIUCDCS-R-78-929, July, 1978.

[3] D. H. Lawrie,
"Recurrence Implimentation Notes - II, 'Constant Coefficient Algorithms'," Dept. of Computer Science, University of Illinois at Urbana, unpublished memo, January, 1978.

[4] S. C. Chen, D. J. Kuck and A. H. Sameh,
"Practical Parallel Band Triangle System Solvers,"
ACM Trans. on Mathematical Software ,
Vol. 4, No. 3, pp. 270-277, September, 1978.

[5] A. H. Sameh and R. P. Brent,
"Solving Triangular Systems on a Parallel Computer,"
SIAM J. Numer. Anal. , Vol. 14, No. 6,
pp. 1101-1113, December. 1977.

[6] F. Panzica,
"Control Structures for Parallel Machines,"
Master's Thesis, University of Illinois at Urbana, in preparation.

# APPENDIX - USER'S GUIDE TO FAPGEN

FAPGEN is a system for the simulation and study of computational algorithms for multiple resource computers. The algorithms are expressed in terms of a user defined set of primitives (MACROS). This section serves as a guide to its use under OS/360. A basic understanding of the theory of operation of FAPGEN is assumed. For this, chapter V should be sufficient. For a more detailed explanation of its structure and methods see the source listing. Preceeding the explanation of FAPGEN is a discusion of the use of PREFAP.

## PREFAP

PREFAP is a procedure for the creation and maintenace of the library of blocks of nodes (MACROS) and the library of tables of node descriptors (TABLES) for later use by FAPGEN. It is designed to run independently of FAPGEN. PREFAP reads the macro definitions and tables of node descriptors from the input file SYSIN, converts them to an internal form, and stores them on disk. A printed output is also generated. To be initiated, PREFAP must be called from a PL/1 procedure as follows:

```
CALL PREFAP ;
```

Macro definitions and the tables are stored in data sets MACROS and TABLES respectively. The index for each is in INDEXS. For each macro definition (MACRO) in the data set MACROS, there must be a corresponding TABLE in the data set TABLES. This does not mean however that for every TABLE in the input file SYSIN there must be a corresponding MACRO in SYSIN, or vice versa. Therefore it is possible to change TABLE definitions in TABLES without disturbing MACROS.

INPUT should appear in file SYSIN in the following form. All commas can be replaced by blanks.

```
MACRO_ACTION = <action> LIST = <list option> ;

   '<macro name>' '<action>'

       <node #>,<# of pred.>,<pred.#1>,...,<last pred.>
                ,<# of succ.>,<succ.#1>,...,<last succ.>
                        .
                        .
                        .
       <last node #>,...
             Ø

   '<next macro name>'...
       .
       .
       .
   '<last macro name>'...
       .
       .
       .
   'ENDM'

TABLE_ACTION = <action> LIST = <list option> ;

   '<table name>' '<action>'

       <node #>,<null op bit>,<res. #>,
              ,<data acces time>,<busy time>
                        .
                        .
                        .
       <last node #>,...
             Ø

   '<next table name>'...
       .
       .
       .
   '<last table name>'...
       .
       .
       .
   'ENDT'
```

| | |
|---|---|
| MACRO_ACTION | <action> can be any one of the following: |
| 'CREATE' | to start a new library or write over an old one. |
| 'UPDATE' | to modify the existing library. |
| 'NO_ACTION' | this is the default. |

| | |
|---|---|
| LIST | <list option>: |
| 'NO' | no output listing. |
| 'SHORT' | prints a list of entries in MACROS. Default. |
| 'LONG' | prints a breakdown by groups of each MACRO being added to MACROS. |
| 'COMPLETE' | prints a breakdown by groups of everything in library. |

Notes: 1. If defaults for both MACRO_ACTION and LIST are to be used then a null field followed by a semicolon must appear in the input stream.

2. If MACRO_ACTION='NO_ACTION' then no input should follow (besides possibly LIST=<list option>) until TABLE_ACTION=<action>.

If MACRO_ACTION = 'CREATE' or 'UPDATE' then one or more MACRO definitions are needed.

| | |
|---|---|
| <macro name> | This is a name consisting of up to 15 legal PL/1 characters. 'ENDM' is used to indicate the end of the MACRO definitions. |
| <action> | Any of the following: |
| 'ADD' | Adds a new entry to MACROS or replaces an old one. If no entry by this name exists in MACROS a new one is created, otherwise the old one is overwritten by the new. |
| 'DELETE' | Deletes an entry from MACROS. |

Note: If ACTION> = 'DELETE' then no input should appear until the next macro name.

If <action> = 'ADD' then one or more of the following node descriptors are needed.

| | |
|---|---|
| <node #> | This is an integer that uniquely defines this node within this MACRO (identified by <macro name>). Maximum of 18 nodes per MACRO. Node number Ø is used to identify the end of this MACRO. |
| <# of pred.> | Maximum number of predecessors is four. |
| <pred. list> | A list of the node numbers of the predecessors. |
| <# of succ.> | maximum number of successors is fourteen. |
| <succ. list> | a list of the node numbers of the successors. If the successor of a node is a node in the next iteration of the graph (i.e., a dependency cycle) then preceed the node number with a minus sign. Within the successor list, all node numbers without minus signs must appear before those with minus signs. |

| | |
|---|---|
| TABLE_ACTION | Same as MACRO_ACTION |
| LIST | Same as before. |

If TABLE_ACTION = 'CREATE' or 'UPDATE' then one or more of the following TABLE definitions are needed.

| | |
|---|---|
| <table name> | This must correspond to a MACRO existing in MACROS. 'ENDT' is used to indicate the end of the TABLE definitions. |
| <action> | Same as before. |

If <action> = 'ADD' then one node descriptor is needed for every node in the coresponding MACRO definition.

| | |
|---|---|
| <node #> | Same node number from the corresponding MACRO. End of TABLE definition is indicated by a Ø node number. |
| <null op bit> | The null operation bit is 'Ø'B or '1'B. For a description of its meaning, see the description of sneak paths in the section on FAPGEN. |
| <resource #> | The box number in the machine. This corresponds to one row in the schedule chart. |
| <data access time> | An integer representing the number of clock cycles needed until the output of this operation is ready. |
| <busy time> | An integer representing the number of clocks needed until the resource can be used again. |

## FAPGEN

After the creation and/or modification of MACROS and TABLES with PREFAP, it is possible to use FAPGEN to simulate the execution of an algorithm or to study individual blocks (MACROS). FAPGEN is composed of a collection of user callable procedures (acually external entry points). These procedures can be called from a PL/1 program and the library containing FAPGEN must be loaded with the user program (driver). This section is divided as follows:

1. Description of the Procedures.
2. The STATS Array.
3. Options and Switches.
4. Example.

1. Description of Procedures

HEADER

Format: CALL HEADER;

Prints a header and the options and switches used. HEADER need not ever be called, but if it is then it must be the first call to FAPGEN.

MAP

Format: CALL MAP ('<macro name>',<# of iterations>);

'<macro name>' must correspond to a name in in MACROS, and be fifteen characters left jusitified, blank fill. <# of iterations> is FIXED BINARY(15).

If this is the first call to MAP with this <macro name> then MAP generates the schedule chart for this block in the form of a HEAD, a BODY window, and a TAIL and stores a record. If it is not then it simply stores a record. If <# of iterations> = 1 there is no HEAD and TAIL, only a BODY.

## IMAP

Format: CALL IMAP ('<macro name>');

This is used to indicate a block which is called with the # of iterations always = 1, such as initialization blocks. Although it usually appears at the beginning or end of and algorithm definition, it can appear anywhere.

## DOLOOP

Format: CALL DOLOOP (<# of iterations>);

<# of iterations> is FIXED BINARY(15).

Used to indicate iteration loops composed of MAP calls or other DOLOOPS. It must be matched by a call to ENDO.

## ENDO

Format: CALL ENDO (<pointer>);

<pointer> is returned and points to a structure which contains time information about the schedule chart for this DOLOOP. (In reality a chart for the whole DOLOOP is not assembled, only its length is computed>). <pointer> must be declared PTR. To receive time information for the entire algorithm, it must be enclosed in a DOLOOP - ENDO pair. Following is the structure that is filled by ENDO and is pointed to by <pointer>. This structure must be declared in the driver in order to access the time information:

```
DCL 1 TIMES BASED(TPT) ,
      2 BOD,                  /* times for BODY
                                 section of the chart */
        (3 TOL,               /*total time */
         3 RES(4)) FIXED BIN(31), /* for each resource */
      2 HEADT,                /* time for the HEAD section */
        (3 TOL,
         3 RES(4)) FIXED BIN(31),
      2 TAILT,                /* time for the TAIL section */
        (3 TOL,
         3 RES(4)) FIXED BIN(31);
```

## DEPEND

Format: CALL DEPEND (<from node>,<to node>);

<from node> is any node in the last iteration of the last block (last call to MAP). <to node> is any node in the first iteration of the next block. Both are FIXED BINARY(15).

DEPEND is used to define data dependencies between adjacent blocks. The node numbers correspond to those in the macro definitions in MACROS. If multiple dependencies exist then several calls to DEPEND are made.

Example:

```
DCL (#2 INIT(2),
     #8 INIT(8),
     #10 INIT(10),
     #11 INIT(11)) FIXED BINARY;
DCL (n1,n2,n3) fixed binary;

CALL MAP ('block1          ',n1);
IF p>2*n THEN DO;
          CALL DEPEND(#10,#2) ; /* two depends */
          CALL DEPEND(#8,#11) ;
          CALL MAP ('block2          ',n2) ;
          END ;
ELSE DO ;
     CALL MAP ('block3          ',n3) ; /* no depends */
```

To declare the dependences between the iterations of a DOLOOP, make the call(s) to DEPEND after the last call to MAP in the DOLOOP and before the call to ENDO. Example:

```
CALL DOLOOP(N1) ; /* loop1 */
        .
        .
        .
   CALL MAP ('block1          ',n2) ;
   CALL DEPEND...  /* for depends between its of the loop */
CALL ENDO(P) ;
     .
     .
     .
CALL DEPEND...  /* for depends between loop1 and loop2 */
CALL DOLOOP...  /* loop2 */
    .
    .
```

## 2. The STATS Array.

The STATS array is a structure used internally by FAPGEN which is also available for the user. Among other things it stores some counts associated with MAP calls. To access STATS the folowing declaration must appear in the driver program:

```
dcl 1 STATS(2) STATIC EXT,
      2 SIZE FIXED BIN, /* total # of macros used */
      2 MAC(200),       /* entries */
        3 KEY CHAR(15), /* macro name */
        3  point ptr,
        3 COUNT,      /* total # of times: */
         (4 STATIC,  /* called */
          4 HITS,    /* "executed" due to "DOLOOPS" */
          4 REPEATS) /* cycled through "microcode" */
    FIXED BIN(31);
```

The first element of the array (i.e., STATS(1)....) is used to store the information about IMAP calls and those MAP calls with the number of iterations = 1. The second element stores information about the remaining MAP calls.

## 3. Options and Switches.

Here is a list of the options and switches (along with their defaults) as they should appear in the input file SYSIN:

```
TRACE = '0'B
OVERLAP_OP = '000'B
COMPRES = '0'B
INIT_MP = '0'B
SNK_PTH(*) = '0'B
F_PRINT = '0'B
T_PRINT = '0'B
O_PRINT = '0'B
DEBUG = '0'B ;
```

SNK_PT, INIT_MP, and COMPRES also declared as external variables in FAPGEN. Therefore it is possible to change their value in the source of the driver.

### TRACE

When TRACE = '1'B a line is printed for each call to DOLOOP, MAP, IMAP, and ENDO, preceded by their respective nest level. For DOLOOP and MAP, the <# of iterations> is also printed. In the case of MAP, the <macro name> is printed.

## OVERLAP_OP

This indicates how sequentially adjacent schedule charts are joined (only the HEADS and TAILS are joined). OVERLAP_OP applies to joining several iterations of the same DOLOOPS also (here, two iterations are joined, then the result is used to compute the times for all the iterations).

```
ØØØ butt
Ø1Ø slide when no depends between
  the two charts in question
Ø11 hop when no depends
1ØØ slide always
1Ø1 hop always
```

## COMPRES

The BODY window is always used to generate the HEAD and TAIL sections of the chart for each MAP call (unless <# of it.s> = 1). With COMPRES = '1'B blank columns are removed from the HEADS and TAILS upon generation.

## INIT_MP

This indicates how the charts of IMAP calls are joined to their adjacent charts. If INIT_MP = '1'B then the charts for all the IMAP calls are hopped into their respective adjacent HEADS or TAILS. If INIT_MP = 'Ø'B then they are butted.

SNK_PTH(*)

This is an array whose size equals the number of resources in the machine. If SNK_PTH(X) = '1'B this indicates that there is a path around resource X in the machine. When scheduling the occurrence of a node in the chart, if there is a path around the box associated with the node and the null_op bit = '1'B (see PREFAP guide), both times associated with that node are set to zero. In this manner it is possible to to simulate any or all of the n**2 connections amoung the n boxes.

F_PRINT

When F_PRINT = '1'B, the schedule charts for the HEAD, one BODY window, and TAIL sections for each MAP call are printed. If <# of it.s> = 1 or the call is to IMAP, then only the BODY is printed. For calls to MAP with <# of it.s> > 1, the initial window size and any updates on the window size is also printed.

T_PRINT

When T_PRINT = '1'B the times that correspond to the schedule charts for MAP and IMAP calls are printed.

## O_PRINT

When O_PRINT = '1'B the amount of overlap is printed as charts are joined. Since the mechanism in ENDO which joins charts works from the bottom of the loop up, this information appears in reverse order. This overlap is with respect to the device busy times (those shown on the chart). If one or more dependencies are present between the two blocks, and if the access times are greater than the busy times, it is possible to get a negative overlap, since dependencies apply to the access times.

## DEBUG

This is used to aid in debugging FAPGEN by printing a message each time a record is pushed on or popped from the stack if DEBUG = '1'B.

## 4. Example

For this example consider Figure A1. Shown is the structure of an algorithm which we wish to simulate. The rectangles are used to represent blocks of nodes (MACROs). We will assume that the data sets MACROS and TABLES were previously created and both contain BLOCK3, BLOCK4, and BLOCK5. This run will add BLOCK1 and BLOCK2 to the libraries and compile and run the FAPGEN driver program EXAMPLE. See Figure A2 for the graph structures of BLOCK1 and BLOCK2.

```
//JOB
/*ID PS=(----,----),NAME='-----'
/*ID CODE=---
/*ID REGION=250K,TIME=(2,0),IOREQ=4000
//* THIS MAY BE MORE THAN SUFFICIENT
//* FAPGEN PROCEDURES ARE IN FAPLIB
// EXEC PLIXLDGO,LIBFILE='USER.P6543.FAPLIB',REGION=250K

/* SOURCE LISTING */

EXAMPLE: PROC OPTIONS(MAIN);

/* DECLARATIONS */

DCL (R,T) FIXED BIN; /* INPUT DATA */
DCL (PT1,PT2) PTR; /* POINTERS TO TIMES */
DCL 1 TIMES BASED(T_PT), /* FILLED IN BY FAPGEN */
       2 HEADT,
         (3 TOL,
          3 RES(4)) FIXED BIN,
       2 BOD,
         (3 TOL,
          3 RES(4)) FIXED BIN,
       2 TAILT,
         (3 TOL,
          3 RES(4)) FIXED BIN;
DCL (HEADER,DOLOOP,IMAP,DEPEND,ENDO) ENTRY;
```

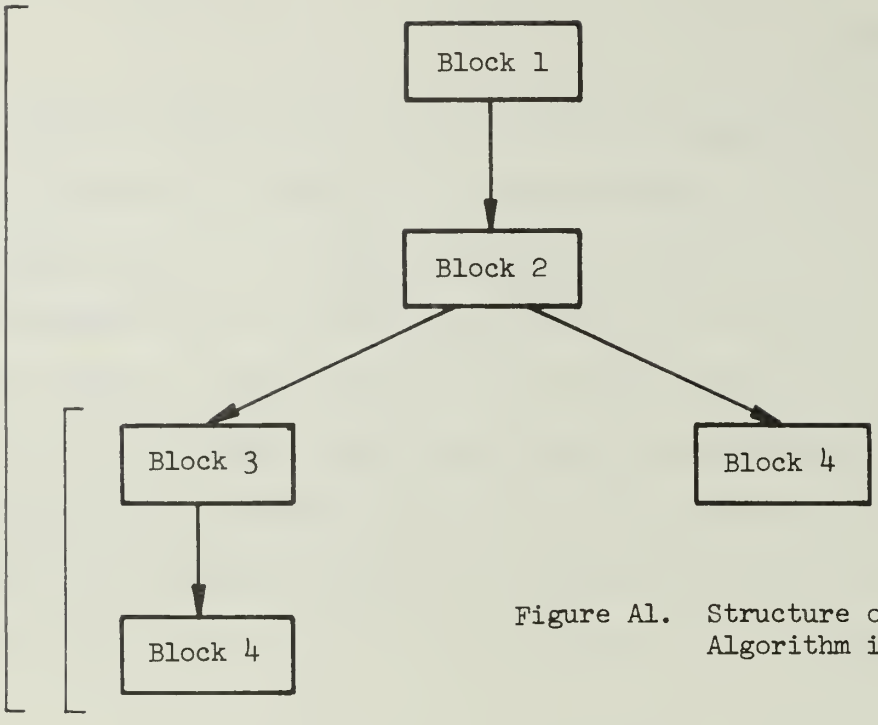



Figure A1. Structure of Algorithm in Example

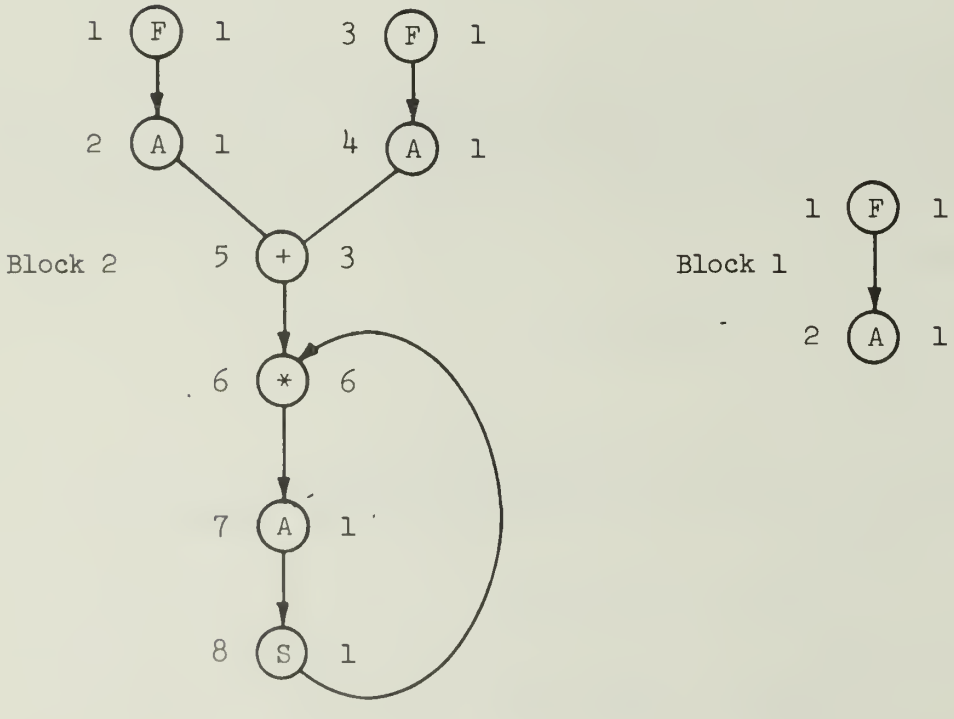



Figure A2. Structure of Blocks

```
DCL (#1 INIT( 1), /* CONSTANTS */
     #2 INIT( 2),
     #6 INIT( 6),
     #8 INIT( 8),
     #10 INIT(10)) FIXED BIN; /* END DECLARATION */

GET SKIP DATA; /* GET SOME INPUT DATA */

CALL PREFAP; /* PROCESS NEW MACROS AND TABLES */

CALL HEADER; /* PRINT HEADER AND OPTIONS */

/* BEGIN ALGORITHM DEFINITION */

CALL DOLOOP(#1); /* SURROUNDS THE ENTIRE ALG.  */
  CALL IMAP('BLOCK1 ');
  CALL DEPEND(#2,#6);
  CALL MAP('BLOCK2 ',#10);

  IF R >= T THEN DO;
     CALL DEPEND(#8,#1);
     CALL DOLOOP(T);
        CALL MAP('BLOCK3 ',R);
        CALL IMAP('BLOCK4 ');
        CALL ENDO(PT2);
     END;

  ELSE CALL MAP('BLOCK5 ',T);

  CALL ENDO(PT2); /* END ALGORITHM DEFINITION */


/* AT THIS POINT THE INFORMATION IN TIMES CAN BE ACESSED */

/* THIS PRINTS THE TOTAL BODY TIME OF THE CHART */

PUT SKIP EDIT (' THE TOTAL BODY TIME = ')(A)
              (T1->TIMES.BOD.TOL)(F(5));

END EXAMPLE; /* END SOURCE */
```

```
//GO.SYSIN DD *

  /* USER INPUT DATA */


R = 10 T = 20;

  /* PREFAP DATA */

MACRO_ACTION = 'UPDATE' ;
'BLOCK1' 'ADD'
1 0 1 2
2 1 1 0
0
'BLOCK2' 'ADD'
1 0 1 2
2 1 1 1 5
3 0 1 4
4 1 3 1 5
5 2 2 4 1 6
6 1 5 1 7
7 1 6 1 8
8 1 7 1 -3
0
'ENDT'
TABLE_ACTION = 'UPDATE' ;
'BLOCK1' 'ADD'
1 '0'B 1 1 1
2 '0'B 2 1 1
0
'BLOCK2' 'ADD'
1 '0'B 1 1 1
2 '0'B 2 1 1
3 '0'B 1 1 1
4 '0'B 2 1 1
5 '0'B 3 3 3
6 '0'B 3 6 6
7 '1'B 4 1 1
8 '0'B 1 1 1
0
'ENDT'

  /* FAPGEN OPTIONS */

TRACE = '1'B
OVERLAP_OP = '100'B
INIT_MP = '1'B;

//GO.TABLES DD DSN=USER.P6543.TABLES
//GO.INDEXS DD DSN=USER.P6543.INDEXS
//GO.MACROS DD DSN=USER.P6543.TABLES
/*
```

| BIBLIOGRAPHIC DATA SHEET | 1. Report No. UIUCDCS-R-79-964 | 2. | 3. Recipient's Accession No. |
|---|---|---|---|

4. Title and Subtitle

SCHEDULING AND SIMULATION OF COMPUTATION GRAPHS FOR MULTIPLE RESOURCE COMPUTERS

5. Report Date: March, 1979

6.


7. Author(s): John Charles Wawrzynek

8. Performing Organization Rept. No. UIUCDCS-R-79-964

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
222 Digital Computer Laboratory
Urbana, Illinois 61801


10. Project/Task/Work Unit No.


11. Contract/Grant No.

US NSF MCS76-81686


12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered

Master's Thesis

14.

15. Supplementary Notes

16. Abstracts


During the design of any new machine there exists a need to provide an accurate simulation of its operation to study its effectiveness. Also of practical importance is the design of a control process which can guarantee a high degree of utilization of the machine resources. In this paper we present a scheme for controlling such a machine and in doing so provide the means for a very precise simulation.

Although this work is directed primarily towards the control and simulation of vector type machines, the theory included in our approach is generalized to include any machine consisting of a collection of interconnected resources. The main theme is to schedule the operations in the resources so as to overlap their activities and keep each as busy as possible throughout the course of a computation.


17. Key Words and Document Analysis. 17a. Descriptors


Microcode
Scheduling
Simulation


17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

| 18. Availability Statement | 19. Security Class (This Report) | 21. No. of Pages |
|---|---|---|
| RELEASE UNLIMITED | UNCLASSIFIED | 100 |
| | 20. Security Class (This Page) UNCLASSIFIED | 22. Price |